A MARSHALL CAVENDISH **21** COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 2 **No 21**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Paul Chave/Image Bank/Don Carroll. Pages 637, 638, 640, Peter Richardson. Pages 642, 645, 647, Steve Bielschowsky, Printers Courtesy of Wilding Office Equipment Ltd. Pages 648/9, Science Photo Library. Pages 650/1, Daily Telegraph. Pages 650, 654, Paul Chave. Pages 657, 658, 660, Mickey Finn. Page 659, Graham Young. Page 663, Paul Chave/Image Bank/Mitchell Funk. Pages 664, 665, 666, Graham Young. Page 667, Kevin O'Keefe. Page 668, Paul Chave/Image Bank/Don Carroll.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN
Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015
New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington
Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

DRAGON/TANDY PROGRAM SQUEEZER

- LOCATING **REM**S AND SPACES
- STORING DATA
- DESIGNATING FLAGS
- MANIPULATING THE STACK
- SHIFTING A PROGRAM

Knock out redundant REMs and superfluous spaces with this machine code routine. It speeds up your BASIC programs and saves memory as well

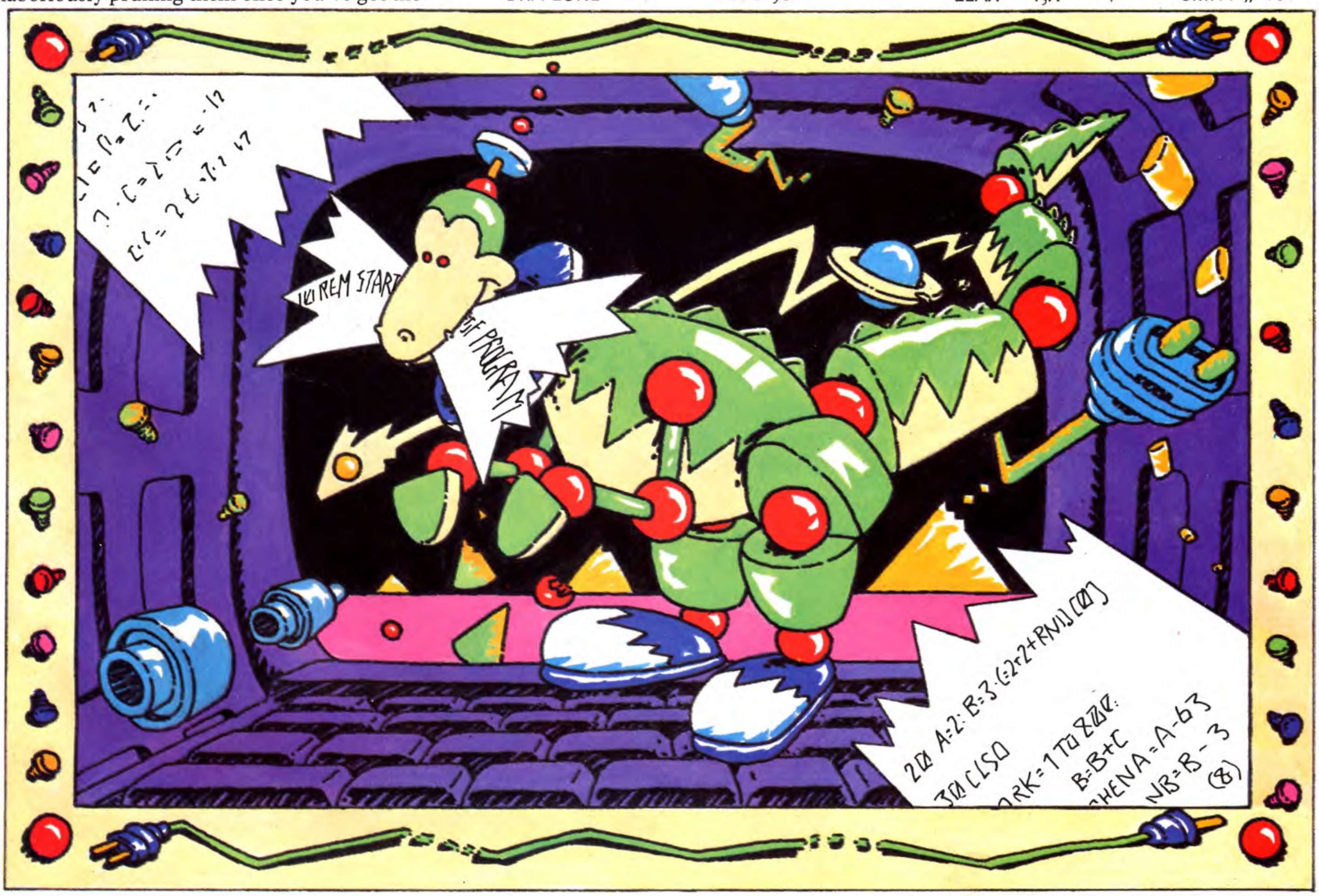
When you write BASIC programs, it is very convenient to put in spaces between instructions, variables and data to make the lines easier to read. And REM statements are often included to make things easier to follow—a great help when you're debugging programs.

The problem is that these spaces and REM statements slow down the programs and take up unnecessary memory space. For maximum efficiency all these unnecessary lines need to be stripped out. But who wants to go through laboriously pruning them once you've got the program RUNning—possibly introducing new errors?

The following machine code routine is a *stripper*. Once you have got your BASIC program RUNing, you run the stripper and it compresses your program for you.

The following is the Dragon/Tandy version of the stripper. For the Tandy, change the instruction picked out in bold to JSR $B4F4. The origin—or the start address, if you're using the machine code monitor—is 30000.

```
     ORG 30000
     LDU 25
     LDD 27
     PSHS D
     BRA LONE
LTWO LDA ,X+
     BNE LTHR
     LDU ,U
LONE CLR ICOM,PCR
     LDD ,U
     BNE LFOU
     LDX 27
     STX 29
     STX 31
     PULS D
     SUBD 27
     JSR $8C37
     RTS
LFOU LEAX 4,U
     BRA LTWO
LTHR CMPA #32
     BNE LFIV
     TST ICOM,PCR
     BNE LTWO
     LDD #1
     PSHS D,X,U
     BSR SHIFT
     PULS D,X,U
     LEAX -1,X
     BRA LTWO
LFIV CMPA #34
     BNE LTWE
     LDB ICOM,PCR
     EORB #1
     STB ICOM,PCR
     BRA LTWO
LTWE CMPA #134
     BNE LSIX
     LDA -2,X
     CMPA #255
     BEQ LTWO
     LDB ICOM,PCR
     EORB #2
     STB ICOM,PCR
     BRA LTWO
LSIX CMPA #130
     BEQ LSEV
     CMPA #131
```



```
        BNE LTWO
LSEV LDA -2,X
        CMPA #255
        BEQ LTWO
        LDD ,U
        LEAX -1,X
        PSHS X,U
        SUBD ,S++
        TFR X,Y
        LDX ,U
        LEAX -1,X
        PSHS D,X
        TFR Y,D
        SUBD 4,S
        CMPB #4
        BNE LNIN
        PULS D
        ADDD #4
        PSHS D
LNIN BSR SHIFT
        PULS D,X,U
        LBRA LONE
ICOM FCB Ø
SHIFT LDD ,U
        BEQ SHTWO
        LDX ,U
        SUBD 2,S
        STD ,U
        TFR X,U
        BRA SHIFT
SHTWO LDD 4,S
        SUBD 2,S
        TFR D,U
        LDX 4,S
SHTHR LDA ,X+
        STA ,U+
        CMPX 27
        BLO SHTHR
        LDD 27
        SUBD 2,S
        STD 27
        RTS
```

HOW IT WORKS

The first instruction LDU 25 loads the U register with the contents of memory locations 25 and 26. These are the systems variables which hold the address of the start of BASIC, and LDD 27 loads the D register with the contents of the systems variable stored in memory locations 27 and 28. These hold a pointer which points to the first free location after the end of the BASIC program. Its value is then stored by PSHS D which pushes it onto the stack.

Later the program is going to work out how many bytes have been saved by the stripping process. So it will need to know where the BASIC program ended before the stripping process began.

The BRA—BRanch Always—instruction then plunges you into the middle of the next subroutine at the label LONE. You'll see why in a moment.

USING FLAGS

CLR ICOM,PCR then CLeaRs the storage area formed by the instruction FCB Ø which follows the label ICOM. The PCR in the clear instruction means Program Counter Relative. Using this instruction means that relative addressing is used, so the program can be relocated in another part of memory if necessary.

FCB means Form Constant Byte. This sets aside one byte for data. Any piece of data can be stored here. But what *is* going to be stored are a couple of indicators which tell the routine whether it is dealing with a string. When it is dealing with a string, it should not strip out spaces. You would not want to strip out spaces between words PRINTed on the screen for example. The bits of the memory location labelled by ICOM is used as a flag, in much the same way as the flags in the condition code register. The flags are set when the routine hits a string and resets them when it leaves, so when it comes to do the actual stripping it can check the flags in ICOM to see whether to proceed.

There is another instruction like FCB. FDB—Form Double Byte—sets aside two bytes for data.

The Ø following the FCB puts ØØ in this byte to start with. (If you put Ø after an FDB, it puts ØØ ØØ into the next two bytes.) And the CLR ICOM,PCR clears this byte again at the beginning of each new line.

UPDATING POINTERS

LDD ,U loads the D register with the contents of the address pointed to by the contents of the U register. In other words, it loads the first two bytes of BASIC into D.

The first two bytes of any BASIC line carry the address of the beginning of the next line of BASIC. If those two bytes contain ØØ ØØ, you've reached the end of the program. BNE—Branch Not Equal to zero—checks for this. If the end of the program has been reached the contents of these locations will not be zero, and the branch is not made.

LDX loads the contents of memory locations 27 and 28 into the X register, Locations 27 and 28 contain the address of the beginning of the variables area, that is, the first free byte after the end of the BASIC program.

This pointer is progressively updated during the machine code routine as the BASIC program is shrunk. STX 29 and STX 31 then puts the same address into 29 and 3Ø, and 31 and 32. These two pointers contain the address of the beginning of the array pointer table and the end of the BASIC area respectively. The variables are defined and arrays are constructed when the BASIC program is RUN. So all these system variables should point to the first free location after the end of the program while you are still rewritting it.

PRINTING ON SCREEN

The value of the end of BASIC pointer for the unstripped program which was pushed onto the stack at the beginning of the routine is then pulled off again. The value of the pointer after the program has been stripped is then taken away from it and the result is stored in the D register.

The JSR $8C37 jumps to a ROM routine which takes the value of the D register, converts it into decimal and prints it on the screen. As you can see, at the end of the program the D register contains the number of bytes saved by the stripper. So when it has finished this stripper program will print out on the screen the number of bytes saved.

This same routine is in a different place in the Tandy's ROM. So it should be called with JSR $B4F4. RTS then ReTurnS to BASIC.

MOVING DOWN THE LINE

If the end of the program has not been reached BNE LFOU sends the program to the next mention of the LFOU label. LEA means Load Effective Address and in this case it is operating with the X register. So LEAX 4,U takes the address from U, adds 4 to it and loads the result into the X register. The first two bytes of any line of BASIC are the start address of the next line, remember, and the next two bytes contain the line number. So this instruction moves the microprocessor to the start of the actual BASIC. The program then branches back to LTWO.



LDA ,X+ loads the accumulator with the first byte of BASIC. The '+' sign means that the X register is incremented after the instruction is executed. This points the address in the register to the next byte of the program, ready to deal with that when the time comes.

BNE LTHR branches to the label LTHR when the contents of the accumulator are not zero. A zero byte marks the end of a line of BASIC. If the end of the line has been reached, the branch is not made and the U register is loaded with the contents of the memory location pointed to by the contents of U, by the instruction LDU ,U.

U was loaded with the system variable pointing to the beginning of BASIC to start with, you'll remember. On the Dragon and Tandy, the first two bytes of each line of BASIC is the address of the beginning of the next line. Each time the program reaches the end of a line of BASIC, the U register is updated this way to point to the beginning of the next line. It then starts again with the clear instruction.

DEALING WITH SPACES

If the end of the line has not been reached the branch is made and CMPA #32 checks to see if the byte loaded in the accumulator is a space. 32 is the ASCII for a space. If the value of the byte is 32 and is a space, CMPA #32 returns Ø so the following branch instruction, BNE LFIV, is not made.

TST ICOM,PCR checks the status of the memory location ICOM. If any of the flags in ICOM are set—in other words, it's not Ø—the routine is in a string, so the space should not be stripped and BNE LTWO branches back to the beginning of the program again. It simply ignores the space. Otherwise, the processor continues on to the next instruction of the routine LDD #1.



USING THE STACK

LDD #1 loads the D register with the number 1, then PSHS D,X,U PuShes the contents of the D, X and U registers onto the Hardware Stack. It may seem a little unnecessary to push the contents of the D register onto the stack as you know it contains 1, but the next instruction BSR SHIFT, Branches to the SHIFT SubRoutine. This subroutine is going to be used to shift the rest of the program back when a space is being taken out. But it is also going to be used to shift the program back when a REM statement is being removed. The value stored on the stack from the D register is the number of bytes the rest of the program must be shifted back and is referred to in the subroutine. In the case of a space, it is 1. But when a REM statement is removed it could be anything up to 255.

A branch to subroutine instruction, BSR, is used here rather than a jump to subroutine, JSR, so that the routine can be relocated anywhere in memory. A branch is a relative jump, so the routine branches a certain number of bytes from instruction. But a JSR was used when the program jumped to the hex-decimal ROM routine, because that jumps to a particular address wherever it is called.

After the shift has been done the values of the D, X and U registers are restored again by PULS D,X,U which PULls the items off the Stack again and returns them to the appropriate registers. You will note that the D, X and U are specified in the same order when you pull them off as when you pushed them on. Bearing in mind how the stack works (see page 237) you would expect the pull order to be the other way round.

But, in fact, it doesn't matter which way you specify them. The microprocessor has its own in-built push and pull order which it will follow, however you specify the registers. The push order is: the low byte of the PC register; the high byte of the PC register; the low byte of the U or S register; the high byte of the U or S register; Y low; Y high; X low; X high; DP; B; A; and CC. The pull order is exactly the opposite—CC first, PC last. You can also see why the two bytes of a 16-bit register don't get switched round when they are pushed onto the stack and pulled off again.

The U and the S registers are the stack pointers to the two stacks—U for the user stack and S for the hardware stack. Obviously you cannot push the value of a stack pointer onto its own stack, nor can you pull an item off the stack and put it in its own stack pointer. That's why push-pull order can specify U or S, and the instruction tells the

processor which stack is being used. PSHS and PULS use the hardware stack and PSHU and PULU use the user stack.

LEAX −1,X decrements the X register—it takes the contents of the X register, subtracts 1 (or adds −1) and loads the result back into X. This is done because the rest of the program has been shifted one byte down memory. X is the pointer to where you are in a line of BASIC. It was pointing to the space which has now been moved. But the next byte of the program has now been moved into the memory location that the space formerly occupied. So to deal with this byte, you have to examine the same memory location all over again, rather than moving onto the next one. When BRA LTWO branches back to the beginning again, the first instruction increments the X register to move onto the next byte, you'll note.

CHECKING FOR STRINGS

If the routine doesn't hit a space with the instruction CMPA #32, the BNE LFIV sends it off to a little routine that checks for quote marks. CMPA #34 compares the byte in the accumulator with the ASCII for a quote.

If a quote mark is found, CMPA #34 will return the value Ø—so the condition of the BNE LTWE will not be fulfilled and LDB ICOM,PCR will load the B register with the contents of the memory location where the command flags are stored.

EORB #1 Exclusively ORs it with 1. The machine code exclusive—or works exactly the same as the exclusive—or logical operator in BASIC (see page 287). STB ICOM,PCR then stores the result back in this command byte effectively setting bit zero as a quote flag.

You can see that if bit zero or ICOM was already set to 1, EORB #1 will give the result Ø, and if the bit zero was Ø, the result is 1. So if bit zero was set, this operation resets it, and if it wasn't set, it sets it.

If you think about how quote marks enclose a string, you will see that a string starts after an odd number of quotes and ends after an even number. By setting and resetting the command flag each time it hits a quote mark, the EOR sets the command flag after an odd number and resets it to Ø after an even number. So when the command flag is set, the routine is in a string, and when it is not set, it's not in a string. This is exactly the condition that is tested for by TST ICOM,PCT and BNE LTWO above.

Once the condition of the quote flag has been altered, BRA LTWO branches back to the beginning of the routine again, ready to deal with the next byte.

But if the byte was not a quote, the BNE LTWE condition would have been fulfilled, and the processor would have branched on to the next check.

CHECK FOR DATA

DATA statements sometimes contain strings so it is as well not to remove spaces from them, too.

CMPA #134 checks to see if the byte in the accumulator is the token for DATA. If it is the condition for BNE LSIX is not fulfilled.

But there is another circumstance in which the token 134 might appear. The function LOG has the two-byte token FF 86 in hex, or 255 followed by 134 in decimal. Space can be taken out of LOGs so it would be as well to check for 255 in the byte before.

Remember that the LDA ,X+ increments the X register after it has been loaded into the accumulator. So the X register is now pointing to the byte past the one being checked. So to look at the byte before the one being checked, the accumulator has to be loaded with the contents of the memory address pointed to by the contents of the X register −2. LDA −2,X does this.

CMPA #255 then checks for 255 in this byte. And BEQ LTWO takes the processor back to the beginning again, if FF is found.

If FF isn't found, the $86 is the token for DATA and a flag in ICOM needs to be set. Obviously, you don't want to set bit zero, because that deals with quotes and you don't want to upset the quote count. So LDB ICOM,PCR loads ICOM into the B register. EORB #2 exclusively-ors the contents with 2 which sets bit one.

CHECKING FOR REMS

CMPA #13Ø checks to see whether the next byte is the token for REM. If it is, BEQ LSEV takes the processor on to the label LSEV. If not, CMPA #131 checks for an apostrophe, which works like a REM in Dragon and Tandy



BASIC. If neither of these tokens is found, BNE LTWO branches to the beginning of the program again to start on the next byte.

But if one of them is found, the program has to check for maths functions prefixed with FF in hex again. This is done by the next three instructions—LDA −2,X, CMPA #255 and BEQ LTWO.

Then the D register is loaded with the contents of the U register which carries the address of the beginning of the next line.

LEAX −1,X loads the X register with the contents of the X register minus one—in other words, it decrements the X register. This steps the pointer in the X register back one place so that it points to the REM or apostrophe token itself. This is the point from which you want to delete. Then PSHS pushes the contents of the X and U registers onto the stack and SUBD ,S++ subtracts the contents of the last two bytes on the stack from the contents of the D register, puts the result in D and increments the stack pointer twice.

You will see from the section on stack handling (above) that the contents of the U register go onto the stack first, followed by the contents of X. SUBD ,S++ subtracts the contents of the last item on the hardware stack, S, from the contents of the D register.

The ++ after the S increments the S register—which is the hardware stack pointer—by two. This counts back up the stack, past the two bytes that carried the contents of X taking it off the stack.

TFR X,Y transfers—that is, copies—the contents of the X register into Y. The Y register is not being used for anything else in the program so here it is used as a temporary store for the pointer normally held in X.

LDX ,U loads the contents of address pointed to by U—the address of the beginning of the next line—into X. LEAX −1,X decrements this so that it points to the end of the current line of BASIC. And PSHS D,X pushes the contents of D and X onto the stack so that they can be used in the shift routine.

TFR Y,D then transfers the contents of Y into D. So the position of the REM token is now in D. SUBD 4,S subtracts from D the contents of the word—that is, the two bytes—that starts five bytes up the stack. (S points to the first byte on the stack. 4,S adds four to S, so it points to the fifth byte.) Looking back you will see that this was the former contents of the U register. And U contains the address of the beginning of the line of BASIC—that is, the one which contains the pointer to the beginning of the next line of BASIC.

In other words, the address of the beginning of the line is subtracted from the address of the REM token. The result gives the number of bytes the REM statement is down the line. What you need to know at this point is whether the REM is at the beginning of the line. If it is, the line number can be removed as well. This is checked for by CMPB #4.

If the REM statement is not at the beginning of the line, BNE LNIN jumps straight to the instruction that calls the SHIFT subroutine. But if the result of the subtraction is 4, the last item pushed onto the stack is pulled off into the D register with PULS D. Four is added to it so that the two-byte line number and the two-byte pointer to the start of the next line are also SHIFTed over. The result is then pushed back onto the stack.

The SHIFT subroutine is called by BSR SHIFT, then the contents of the registers are restored by pulling them off the stack again. LBRA LONE branches back to the beginning of the program again.

A Long BRAnch is used here because the jump is more than 128 bytes.

THE SHIFT ROUTINE

The LDD ,U loads the D register with the contents of the address pointed to by U—in other words, D carries the address of the beginning of the next line. If it's zero, you've reached the end of the program and BEQ SHTWO branches out of the loop.

If not, the same thing is loaded into the X register by LDX ,U. The address is in D—the old address of the beginning of the next line—has the two-byte item that starts three bytes up the stack from it, by SUBD 2,S. Remember that the processor is now in a subroutine. So the return address of the subroutine has now been added to the stack pointer.

The result of this subtraction is where the next line should begin once the shift has been made. This is stored back in the memory location pointed to by U, which is the two-bytes at the beginning of the current line of BASIC which stores the beginning of the next line. In other words, the pointer is being up dated. The value of X, which carries the address of the old start of the next line, is then transferred into U and BRA SHIFT takes the processor back to the beginning of the SHIFT routine again. The only way out of this loop is the BEQ which only operates when the end of the program has been reached.

Once that has been completed, the BASIC program can be shifted. LDD 4,S loads D with what was the contents of X on the stack, that is the address of the end of the current line of BASIC if a REM is being dealt with, or the current position in the line if a space is being stripped. SUBD 2,S then takes the number of bytes the program has to be shifted away.

The result is the position that the next byte of the BASIC program has to be moved back to, and it is transferred into U by TFR D,U.

LDX 4,S loads the X register up with the former contents of X from the stack again. So the old address of the next byte of the BASIC program is in X and the address it has to be shifted to.

LDA ,X+ loads the byte pointed to by X into the A register and increments the X register. STA ,U+ stores it in the address pointed to by U and increments the U register. So the next byte of the old, unstripped program is moved into the next position of the new, stripped program. And the two pointers in X and U are incremented to point to the next byte and next position.

CMPX 27 compares the contents of X with the contents of the system variable in memory locations 27 and 28. This points to the beginning of the variables area which comes directly after the end of the BASIC program area. The instruction checks to see if the end of the program has been reached. If it hasn't, the BLO—Branch if LOwer—instruction sends it back to the label SHTHR to do it again.

When all the program has been shifted, the processor moves on to LDD 27. This loads the end of program pointer into the D register. SUB 2,S takes the number of bytes the program has been shifted off again. And STD 27 stores the new, updated pointer back in 27 and 28. RTS returns to the main routine.

HOW TO USE IT

You can assemble this program anywhere you like in memory—assuming you have CLEARed the appropriate space—by changing the ORG. Once you've assembled it, enter NEW to get rid of the assembler and feed in the BASIC program you want to strip. Then you run the stripper with the instructions:

```
DEF USRØ=start address
```

where the start address is the origin, and:

```
PRINT USRØ(Ø)
```

Be careful though. You will not be able to edit your program after it has been stripped so make sure it is RUNning properly first. And watch out for GOTOs and GOSUBs which send the computer to REM.

To SAVE the stripper in machine code use:

```
CSAVEM"STRIPPER"start address, end address, start address
```

To LOAD the stripper back from tape use:

```
CLOADM"STRIPPER"
```

PLUG IN TO A PRINTER

Unlike a typewriter, a printer isn't ready the minute you plug it in. Since it is controlled by the computer, you have to set things up so that they understand each other

With so many different types of printer to choose from, and so many different features on offer, buying one can be a time-consuming business. The article on pages 225 to 229 explains the various types, but you still have to decide what is best for you, your computer and your pocket. And in some cases, you will have to buy special equipment in order to attach the printer to your computer.

So it can be disappointing to discover that you may have to spend as much time again getting your new peripheral plugged in and working, after you've got it home. In some cases, you can spend ages trying to find out what the control and escape codes are, how to send them to the printer, and perhaps most irritating of all, how to get hold of all those special functions that made you go out and buy it in the first place. But this guide should help you avoid some difficulties.

SETTING UP

The function of this article is to show you what to do when you have already unpacked your printer and fitted a plug to the mains lead (this doesn't apply to those printers powered from the computer, which can not—and must not—be plugged into the mains). However don't plug in the printer yet. There are one or two things you must do first, and with many printers the most important of these is removing the shipping screws.

If the print mechanism slides around in transit, it can be severely damaged. So shipping screws are usually put in underneath the printer to stop the mechanism from moving. A few printers (Sinclair's ZX is a notable example) have no transit protection at all—so check the maker's instruction manual carefully. Generally there are only two or three screws, though you may find bits of sticky tape and elastic bands assisting them in their job. Remove all of them—the last thing you need while printing is sticky tape or elastic gumming up the works. Keep the screws and packing handy so that you can replace them if the printer is moved in the future.

It is now safe to connect the printer to the mains (or your computer if it gets its power from there), and switch it on to check that it's working.

Most printers have a little red or green



light to let you know that they are powered up. Even if there is no light you will probably see the printhead move a little as it adjusts itself. Many printers will also give you a wailing noise, a bleep, or some other complaint, if the paper hasn't yet been fitted. If your printer does not work for some reason, try the obvious before taking it back to the shop to complain. It might be something as simple as one of the wires in the plug which has slipped loose. When you've checked that the printer works, switch it off again as you now need to fit the paper (and ribbon if one is required).

PAPER

Your printer will probably use one of two types of paper—either in rolls (like the ZX printer or Commodore plotter) or in folded sheets, perforated along the edges (like Epsons, Seikoshas and most independent printers). The more versatile machines, for example the Epsons, can take both fanfolded paper or separate sheets—a feature that can be useful for letter writing. Electrostatic and thermal printers require special paper, and this is usually machine-specific—for example, you can't use ZX printer paper in any other printer. Have a look at page 228 for more information on the types of paper used by printers.

If you have a choice of papers, make sure you don't choose one that is too narrow or too wide. When fitting it make sure that the paper is neither crumpled nor stretched too taut—the wrong adjustment can give you jammed or torn printouts.

SELF TEST

After making sure that the printer is ready, it's worth trying the self test option, if your printer has one. Many printers will print out a list of their complete character set on command. This serves two purposes; it shows you that the machine is functioning, independently of the computer, and also demonstrates the range of characters that are available. One way to initiate the self test, on the Epson, is by holding down the *form feed* or *line feed* buttons while the power is switched on. On some machines there is a special switch for the purpose.

RIBBONS

Unless yours is an electrostatic or thermal printer, it will probably need a ribbon. Most ribbons now come in cartridge form, and these are not hard to fit if you look at the diagram in your printer manual. Less simple are the spool-to-spool type used in some Commodore printers, which fit in like a typewriter ribbon. Fitting any type of ribbon is only a matter of practice—after two or three goes you'll find that the ink stays on the ribbon, rather than finding its way onto your fingers.

CONNECTING TO THE COMPUTER

The last, and most important, link in the chain is attaching the printer to the computer. In many cases this is as simple as plugging the lead into the right socket, which is what you do with the ZX and Commodore dedicated printers. Often, as on the Spectrum, there is only one place that this can plug in. On the Commodore computers you have to be careful if you are using disks as well as a printer—in this case the printer plugs into the disk drive, which then plugs into the computer. This system is known as a daisy chain and must be used with some care—see Troubleshooter.

It should be no more difficult to plug in a printer from an independent manufacturer. If you have a Dragon, you should have a printer with a Centronics interface (the more popular printers, like Epsons, Seikoshas, Canons and most daisywheels now have this as standard) and you'll need a Centronics lead to go from the printer to the side of your Dragon. The BBC works with printers that use either the Centronics or RS232 leads (it actually has an RS423 port, which is RS232 with variable speed control), while the Tandy needs an RS232 printer.

Make sure when plugging in any leads that they are plugged in the right way up. The standard Centronics and RS232 plugs make you do this anyway, but be careful with the



socket underneath the BBC. Be sure to align up the notches on both lead and socket.

If you have an independently made printer for a Spectrum or Commodore, then you will also need an interface. The Electron, too, needs an interface before it can be connected to any sort of printer, as there is no printer port provided at all. An interface is slightly more complicated to use because you may have to load in a program before the interface will work. However, once you have one you can attach just about any printer to your computer. There are even interfaces that allow you to use a ZX printer with a BBC, or Commodore printer with a Spectrum. Many interfaces are available, and it's worth shopping around if you need one, as you need to make sure you see it working before you buy.

CONTROL COMMANDS

The most important thing that a printer has to do is to print, but it must print when you want and what you want. Anything to be printed must be sent down the line from the computer before it can appear on paper. The commands used to control the printer vary from computer to computer but mostly you can do all you need with one or two simple instructions.

The Spectrum and ZX81 do all of the talking to the ZX printer with simple BASIC commands which make it one of the easiest combinations to use. The command LPRINT has a similar effect as PRINT, but instead of the output going to the screen it goes to the printer. With the Spectrum, you do not have the same freedom to specify the type of printing as you do with the ordinary PRINT. You can use INVERSE with the LPRINT command, but PAPER, INK, OVER, FLASH and BRIGHT do not work—indeed you would hardly expect them to with a black and white (or black and silver) printer.

In the same way as the PRINT command is replaced by LPRINT, the LIST command is replaced by LLIST, which lists the program currently in memory onto the printer. Note that while you have more than one screenful of program, entering LIST will produce a 'scroll?' message, on the other hand LLIST will generate a continuous printout regardless of the program lengths.

The third BASIC command that you can use with the ZX printer is COPY. This makes a copy of the picture or whatever is on the screen, a process sometimes called a *screen-dump*. When the printer is used like this to produce dots rather than letters, it is known as graphics printing. You can use COPY to get a hard copy of the results from programs, regardless of whether these are text or graphics.

While working, the printer can be stopped at any point by pressing the BREAK sequence ([CAPS SHIFT] and [SPACE]). But unlike the normal procedure, you cannot continue after this—a printout must be restarted from the beginning.

If you're not using the ZX printer, but an independent printer, then the chances are that you will be using an interface. Most of these are controlled by the same Spectrum commands, but you end up with a higher quality printout, as well as the chance to use wider paper, or faster printouts, depending on how much you've paid for the printer/interface combination. Obtaining screen-dumps from an independent printer can be more complicated than listing programs. A future article in *INPUT* will explain how you can do this.

Getting printouts from the Commodores is more complicated than on machines like the Spectrum, because of the way Commodore peripherals are arranged in a daisy chain. The printer could be one of several devices on the same line so it has to be addressed separately.

Each of the Commodore's peripherals has a different address; 1 is the cassette recorder, 2 is a modem, 3 is the screen, 6 is the plotter and 8 to 11 are disks of one sort or another. Addresses 4 and 5 are reserved for the printer. Most printers use 4 but some have a switch at the back so they can be set to either 4 or 5.

It is not possible to send information directly to the printer. First you have to associate a logical file with the printer and then send the information to that. This is not difficult to do. The first command:

```
OPEN 1,4
```

associates logical file 1 with a device 4—the printer. The second command:

```
CMD 1
```

then directs all output—that is, all normal PRINT or LIST commands—to file 1 rather than the screen. You can use any file number from 1 to 255 but numbers over 128 are really designed for other purposes—such as for use with non-Commodore printers.

If you want to send just a small amount of data to the printer you can use PRINT# 1 instead of CMD1 followed by the data you want printed. For example:

```
PRINT#1,"TITLE PAGE"
```

sends the words TITLE PAGE to the printer.

When you've finished printing enter:

```
PRINT#1 : CLOSE 1
```

to close the file and redirect information to the screen.

The plotter is used in a similar way, by OPENing a logical file to device number 6. If you have both a printer and a plotter connected you could set up two logical files:

```
OPEN 1,4: OPEN 2,6
```

and then use either CMD or PRINT# to direct data to the correct file.

You can include codes in the messages that you send, to access special features of the printer. To do this you simply include the CHR$ of one of the following numbers in the PRINT statement:

10	Line feed
13	[RETURN], gives automatic line feed
14	Begin printing in double width mode
15	Stop printing in double width mode
18	Begin printing in reverse character mode
146	Stop printing in reverse character mode
17	Switch to upper and lower case graphics
145	Switch back to upper case and graphics character set

Q+A

Why does PRINT# give error messages on the Commodore machines?

PRINT# is a useful command for directing information to a printer, but in some cases it can be the cause of annoying errors that are hard to spot.

If you get syntax error messages associated with lines containing PRINT#, always suspect this first. A common error is leaving a space before #, but there may be another cause.

The Commodores allow you to use ? as a shorthand way to enter PRINT. But this is not permissible before a #. Here the abbreviation is [P] [SHIFT] [R] (type [R] with [SHIFT] held down). It can be maddeningly difficult to spot the problem, as once a ? has been entered, the Commodore's program listings display the word PRINT in full, and not the abbreviations. The only way to be sure is either to edit in the new letters over the old ones, or to retype the line, whichever is quicker. And, of course, don't use the ? when you next enter PRINT#.


brother

For example this line:

```
10 OPEN 1,4 : CMD 1 : PRINT
   CHR$(14);"DOUBLE"
```

would print the word DOUBLE in double width on the printer. See the section on Escape codes, below, for more on these control characters.

Like the Spectrum, the Commodores use an unusual printer connection, but you can buy a Centronics interface which will allow you to use almost any printer, although you usually need special software which you have to RUN first to enable the Centronics printer. The disadvantage with this is that you will no longer be able to print the graphics characters and symbols embedded in your programs. One answer to this is to adopt the idea of using CHR$'s of characters rather than the graphics themselves—this can save a lot of confusion, although you then need to learn the CHR$ numbers!

Using a printer with the Acorn computers is extremely easy. You can switch output to the printer by pressing CTRL and B together, and you can turn it off by pressing CTRL and C. The same thing can be done from within a program using the VDU commands VDU2 and VDU3 respectively. The VDU2 (or CTRL and B) actually sends the output to both the printer and the screen. If you want to turn off output to the screen use:

```
*FX3,2
```

and use *FX3,0 to return to normal.

The Acorn computers assume that your printer uses a Centronics (or parallel) connection. If you have a serial printer, connected via the serial port, you need to use two *FX calls. These are:

```
*FX 5,2 and *FX 8,N
```

The first of these simply tells the computer that you are using the serial interface, while the second tells it which baud rate the printer uses. The number N ranges from 1 to 8 to select different baud rates from 75 to 19200 baud. Your printer manual should tell you which rate to use.

If you want to change back to a Centronics or parallel printer use *FX 5,1.

You can print from your Dragon or Tandy computer very easily, using a few simple BASIC commands.

To list a program, simply type LLIST (followed by ENTER). Just to print a word, or text, use the following command:

```
PRINT# -2, "TEXT"
```

This is actually telling the computer to send its output, in this case the word TEXT, to the peripheral −2, the number for printers.

You should note that after the −2 in the command above, there is a comma. This tells the computer to expect the text you want printed.

Both the Dragon and Tandy have built-in interfaces. The Dragon comes with a Centronics interface, while the Tandy has an RS232. Both of these are home computing 'standards' and so you should have no trouble connecting your printer to your computer.

DIP SWITCHES

As you delve deeper into printing you will come across two other confusing features—dip switches and Escape sequences.

Inside most printers and some other electrical equipment, you will find banks of tiny switches—the dip switches. These can be used to alter the state of the printer on powering up—the *default* state. Some things that are commonly set with dip switches are the line lengths, line spacings, carriage returns, typefaces and page lengths. In addition, Japanese printers often allow you to select the international character set that you want (for example, the pound sign as opposed to the dollar, or accents for certain characters).

To get at the dip switches consult the printer manual—they are usually reached by taking off the printer case, but don't remove it until you have switched off the power. Once you have found the switches you can change the settings by pushing them gently with your finger or a small screwdriver. Again, the manual should have a diagram of the switches, and the settings that will give you what you want can then be selected. Test the new version with the self test option. Any formatting options that you have selected can be checked with the letter writing program given on pages 124 to 128.

Some printers will allow you to select certain features with either dip switches or by sending control or Escape sequences to the printer—especially features such as line width, typeface, carriage return and justification (alignment to a margin).

ESCAPE CODES

It is possible to send control codes to some printers, as well as the data you want printed—the Commodore is one example. These codes are often known as Escape sequences, as many printers expect you to put the code for Escape, CHR$(27), before the control code. This lets the printer know that what is about to come is a message, and not a character to be printed.

Each printer has its own specific Escape sequences, although those used on the Epson are rapidly becoming a standard on printers which offer similar facilities. On the more versatile printers there is often a huge range of these Escape sequences, allowing you to set any of the special features. Some Epson compatible printers have more than 40, as well as 12 dip switches inside. The Escape sequences can be sent in normal PRINT statements—for example the following, sent to an Epson, would print the word INPUT in emphasized mode, assuming the printer has already been sent the correct commands to turn it on (if these are necessary):

```
LPRINT CHR$ 27;"E";"INPUT"
```

```
PRINT#1,CHR$(27);"E";"INPUT"
```

```
VDU 1,27,1,69:PRINT"INPUT"
```

```
PRINT# -2,CHR$(27);"E";"INPUT"
```

But don't forget to turn back to normal mode afterwards:

```
LPRINT CHR$ 27;"F"
```

```
PRINT#1,CHR$(27);"F"
```

```
VDU1,27,1,70
```

```
PRINT# -2,CHR$(27);"F"
```

Remember, these codes are for the Epson and Epson-compatible printers only. Other printers use different codes.

The Acorn codes need some explanation, as the codes are sent by means of VDU commands rather than CHR$'s. The code numbers themselves are the same, although ASCII codes are used rather than letters such as E and F. Also each code number needs to be preceded by the number 1 which means 'send the next character to the printer only'.

EPSON CODES

Among the effects you can achieve with the Epson-type escape codes are italic print, underlined print, bold print, subscript and

TROUBLESHOOTER

- When used with more than one peripheral, Commodore machines are sensitive to the order in which they are powered up—and in some cases may even be damaged by being switched on in the wrong order.
- One common example of connecting more than one peripheral is if you use both a printer and a disk drive in daisy chain form, as described in the main text.
- In these circumstances, you should always power up in this order: **computer, disk drive, printer**. This is the latest recommendation from Commodore and differs from advice given in some of the manuals. It *may* be safe to switch on a printer first, but if you have a Commodore 1526 printer, switching this on before the computer will result in destruction of the computer's main chip—an expensive way to find out the importance of the correct sequence.

superscript. Of course, the actual changes you can get vary from printer to printer, but here are a few examples which work on most Epson-compatible printers.

ESC 4	turns on alternative character set (usually italics)
ESC 5	returns to the standard set.
ESC –	(a minus sign) turns underline on and off. It must be followed by ;CHR$(1) or by ;CHR$(49) to turn it on, and is turned off by ;CHR$(Ø) or ;CHR$(48), depending on whether CHR$(1) or CHR$(49) was used to turn the mode on.
ESC G	turns on double-strike mode.
ESC H	turns off double-strike mode.
ESC S	when followed by ;CHR$(Ø); this prints in superscript mode; when followed by CHR$(1) it prints in subscript mode.
ESC T	returns the printer to normal mode after either subscript or superscript mode.
ESC @	initializes the printer and resets all other commands.

SEIKOSHA CODES

The Seikosha-compatible printers, for the most part, do not have as large a range of type styles, but most of them can support elongated type. To turn on the elongated mode use:

```
LPRINT CHR$ 14;"INPUT"
```

```
PRINT #1,CHR$(14);"INPUT"
```

```
VDU 1, 14:PRINT"INPUT"
```

```
PRINT # –2,CHR$(14);"INPUT"
```

You can turn off this mode using:

```
LPRINT CHR$15
```

```
PRINT #1,CHR$(15)
```

```
VDU1,15
```

```
PRINT # –2,CHR$(15)
```

With all printers you can use the ESCAPE code as you would any other character: by putting them after the command which your computer uses to send output to the printer (LPRINT, or PRINT # –2, or whatever), and separating them with semi-colons, as you would any other CHR$s.

Some printers, though not all, require you to put the ASCII code for ESCAPE between every control code. Your printer manual should say whether or not you need to do this.

NEW TRENDS

Escape sequences will become even more important as printers get cheaper and more sophisticated. Already there are dot matrix printers whose quality is almost indistinguishable from daisywheels; some of these have built-in features usually associated with typesetting, like proportional spacing and automatic left or right justification.



DECODING YOUR EPIC ADVENTURE

Complete your text compressor by adding the decode routine. And if you don't have a suitable assembler, there's also a ready-to-use hex listing of the whole compressor

The first part of this article explained how it is possible to encode text in binary numbers which take up less memory space than the usual ASCII coding system. And the assembly language listings on pages 630 to 636 contained a program to do just that.

But coding all your text in this way is no use if the words remain locked in a mass of binary numbers. So the counterpart of the encode routine is a decoder. The assembly listings which follow add the new section which will recover the message for you.

There is also a complete hex listing of the whole text compressor, which you can use if you don't have a suitable assembler. Whether you use this or the assembly language version, when you have completed the program, SAVE the code on tape. Next time, you'll see how to use the machine code within an adventure game, and how to develop your games in conjunction with the text compressor.

HOW IT WORKS

Starting at the beginning of the binary coded material, the machine takes one bit at a time, gradually building up a longer and longer binary number. After each new bit has been added, the number that has so far been built up is compared to the codes in the table, looking for a unique match. If no match is found, or if there is more than one possible match, the machine adds the next bit to the number. It stops building up the number when it finds one single matching binary code. If the machine starts reading the code at the correct place, it will always find a single matching code.

Having made the match, the character is put in a string, ready to be printed out in BASIC. The existing binary code is forgotten, and the next binary number is built up in the same step-by-step fashion.

The computer knows that the piece of text is complete when it finds the 'message ends' code, it's then up to the accompanying BASIC program—given in the final part of this article—to display the text on screen.

If you have decided to use the assembly listing, LOAD in the existing source code and



- ASSEMBLY LANGUAGE LISTING FOR DECODER
- HOW DECODERS WORK
- MATCHING BINARY CODED MATERIAL
- CONVERTING CODES BACK TO BASIC
- HEX LOADING PROGRAM FOR HEX DECODER
- STORING THE DECODER



continue adding to it—the following section is the decoder. When you have finished entering the listing, assemble the complete program and SAVE it on tape—see your assembler's manual for how to do this.

```
DFIND   PUSH  IX
        PUSH  IY
        CALL  ESETUP
        LD    C,(IY+8)
        LD    B,(IY+9)
        ADD   IY,BC
        LD    C,(HL)
        INC   HL
        LD    B,(HL)
        INC   HL
        PUSH  HL
        ADD   HL,BC
        LD    (DCTEST+1),HL
        LD    A,7
        LD    (BITTY+1),A
        LD    HL,CSP
        LD    (CHAR+2),HL
        POP   HL
        LD    A,H
        JR    PUSHL
DCHASH  LD    A,9
DCLOOP  CP    #20
        CALL  NC,DECODE
        AND   A
        JR    Z,DCHASH
        POP   HL
        LD    (HL),A
        INC   HL
PUSHL   PUSH  HL
DCTEST  LD    BC,#FFFF
        SBC   HL,BC
        JR    NZ,DCLOOP
        POP   HL
        POP   IY
        POP   IX
        RET
DECODE  LD    IX,LO
BITTY   LD    DE,#00FF
        ADD   IX,DE
        LD    A,(IY)
        XOR   (IY+1)
        AND   (IX+1)
        XOR   (IY+1)
        LD    B,E
        INC   B
BRAN    RRCA
        DJNZ  BRAN
        RLA
        DEC   E
        DEC   E
CHAR    LD    IX,#FFFF
        JR    C,LONGD
        RLA
        JR    C,THREE
        LD    A,(IX+FIRST-CODE)
        JR    FOUND
THREE   DEC   E
        RLA
        LD    A,(IX+SECOND-CODE)
        JR    NC,FOUND
        XOR   A
        JR    FOUND
LONGD   RRA
        DEC   E
        DEC   E
        DEC   E
        CP    255+CEIGHT-CLAST
        JR    NC,EIG
        ADD   A,255+CEIGHT-CLAST
        SRA   A
        OR    #80
        CP    255+CSEVEN-CLAST
        JR    NC,SEV
        ADD   A,255+CSEVEN-CLAST
        SRA   A
        CP    255+CSIX-CLAST
        JR    NC,SIX
        ADD   A,255+CSIX-CLAST
        SRA   A
        JR    FIV
EIG     DEC   E
SEV     DEC   E
SIX     DEC   E
FIV     ADD   A,CLAST+1-CODE
FOUND   LD    HL,CODE
        LD    C,A
        ADD   HL,BC
        LD    A,E
        CP    %10000000
        JR    C,NOCHAN
        ADD   A,8
        INC   IY
NOCHAN  LD    (BITTY+1),A
        LD    A,(IX)
        CP    "_"
        LD    A,(HL)
```

```
         JR     NZ,NOTLWR
         LD     IX,CPUN
         LD     (CHAR+2),IX
         SUB    #20
         RET
NOTLWR   CP     "↑"
         JR     NZ,NOTUPP
         CALL   DECODE
         JR     NZ,NOTUPP
         ADD    A,#20
NOTUPP   LD     (CHAR+2),HL
         CP     "_"
         JP     Z,DECODE
         CP     "□"
         RET    Z
         XOR    #20
         RET
```

If you don't have a commercial assembler for your Spectrum, you'll need to use the hex listing that follows. But before the hex can be entered, you'll need to enter these commands directly:

```
CLEAR 64599
NEW
```

Now type in this hex loading program. SAVE it on tape because you'll need it next week for entering the decoder.

```
10 POKE 23658,8
20 INPUT "START ADDRESS?□";SA
30 INPUT LINE D$
40 IF D$="" THEN GOTO 30
50 IF D$(1)="#" THEN STOP
60 IF LEN D$=1 THEN GOTO 30
70 LET S$=D$(1 TO 2)
80 FOR N=1 TO 2
90 IF S$(N)>"9" THEN LET S$(N)
   =CHR$ (CODE S$(N)-7)
100 IF S$(N)>"?" OR S$(N)<"0"
   THEN PRINT FLASH 1;"INVALID□
   CHARACTER":GOTO 30
110 NEXT N
120 POKE SA, (CODE S$(1)-48)*16
   +CODE S$(2)-48
130 PRINT SA,D$( TO 2)
140 LET D$=D$(3 TO )
150 LET SA=SA+1
160 GOTO 40
```

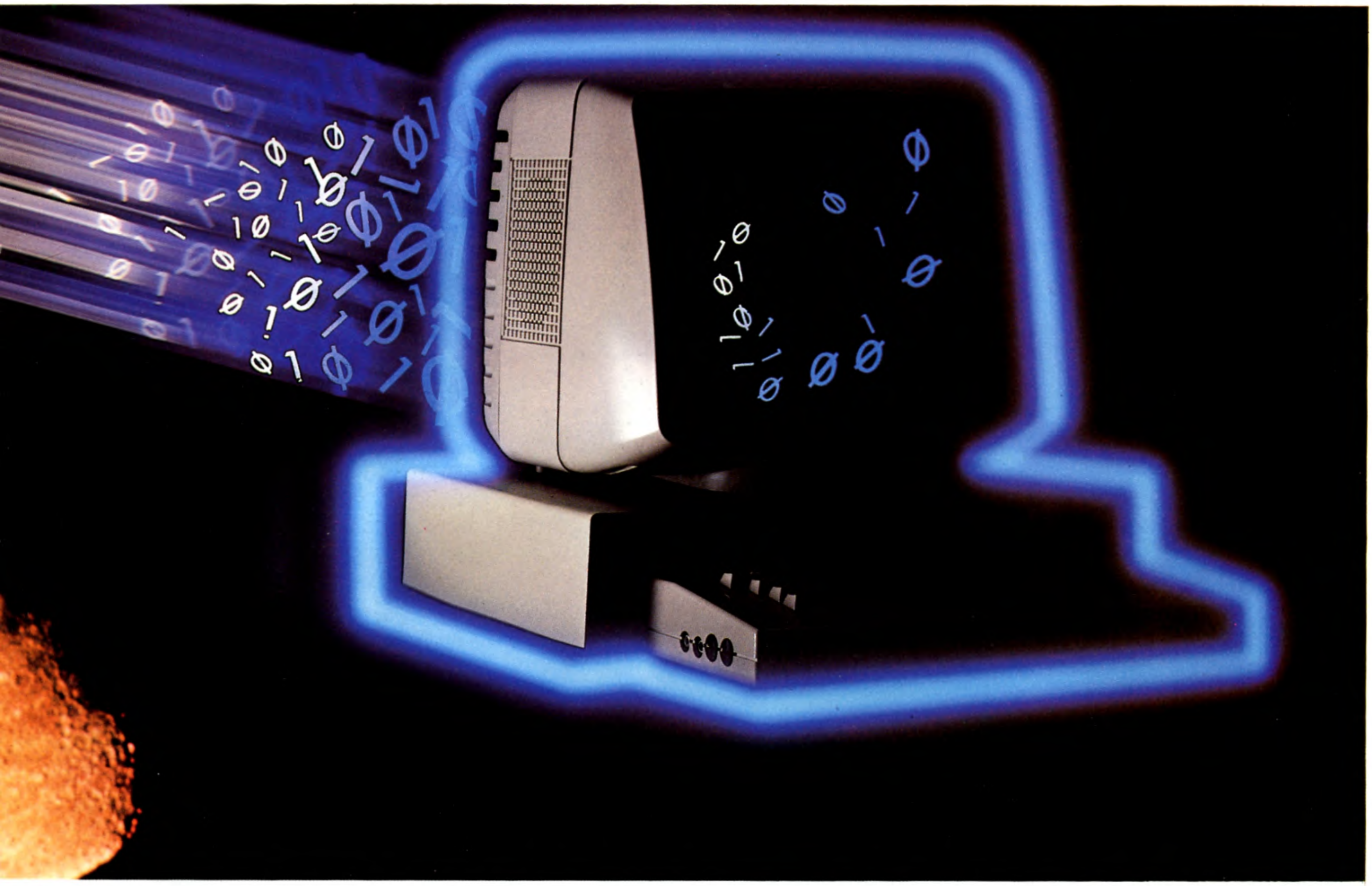
RUN the program, and enter the start address—64600. Now feed in all the hex numbers, but not the spaces! Then when you've finished, the machine code program will have to be SAVEd on tape. Type SAVE "txt comp" CODE 64600,684 and the machine code will be SAVEd.

```
21 0B 00 22 67 FC C9 DD E5 FD E5 CD DC
FD 01 FF FF FD 71 08 FD 70 09 FD E5 FD 09
4E 23 46 E5 23 09 22 8D FC 3E 07 32 E6 FC
21 69 FD 22 E3 FC E1 23 E5 7E A7 01 FF FF
ED 42 28 07 CD B5 FC C6 00 28 ED E1 3E 00
CD B5 FC FD 23 FD E5 E1 C1 3A E6 FC C6 F9
ED 42 22 67 FC FD E1 DD E1 C9 06 20 FE 40
38 02 EE 20 4F 21 88 FD 96 28 1C FE 20 20
0B 3E 7E E5 C5 CD B5 FC C1 E1 18 0D FE E0
20 04 3E 7F 18 EF 79 2B 10 E2 C9 78 DD 21
FF FF 1E FF 1D 1D 3D 28 13 DD BE 20 20 04
3E 00 18 31 1D DD BE 41 20 09 3E 40 18 27
3E 60 1D 18 22 1D 1D 1D 1D 1D C6 E0 FE F8
30 17 1C CB 27 C6 08 FE F0 30 0E 1C CB 27
C6 10 FE C8 30 05 1C CB 27 C6 38 4F 7E FE
5E 28 0D DD 7E 00 FE 5F 20 03 21 89 FD 22
E3 FC 79 ED 4B E6 FC 41 04 07 10 FD DD 21
CB FD DD 09 47 DD A6 01 FD B6 00 FD 77
00 78 DD A6 09 FD 77 01 7B FE 80 38 04 C6
08 FD 23 32 E6 FC 3E 00 C9 20 41 53 4F 54
52 49 4C 45 43 5E 5F 55 4D 50 57 59 4E 42
47 44 46 56 48 4B 51 58 4A 5A 5B 5C 5D 0A
11 04 11 17 08 11 08 05 17 02 07 11 08 03
01 01 14 08 08 08 04 08 08 08 0C 0E 0C 08
0B 0B 00 0B 04 04 08 15 06 03 04 06 14
03 04 11 02 01 08 17 03 04 07 17 06 03 06
01 06 1A 04 03 06 0A 1E 0B 0A 00 01 03
07 0F 1F 3F 7F FF FE FC F8 F0 E0 C0 80 00
2A 4B 5C 16 00 0E 02 7E FE E0 30 12 FE C0
30 26 FE A0 30 0F FE 80 30 28 FE 60 30 0D
18 10 1E 13 19 18 E4 23 7E FE E0 38 FA 1E
06 19 18 D9 FE 5A 20 04 E5 0D 28 16 23 5E
23 56 23 19 16 00 18 C7 FE 9A 20 F2 E5 FD
E1 0D 28 02 18 EA E1 23 C9 DD E5 FD E5 CD
DC FD FD 4E 08 FD 46 09 FD 09 4E 23 46 23
E5 09 22 61 FE 3E 07 32 72 FE 21 69 FD 22
8C FE E1 7C 18 0D 3E 09 FE 20 D4 6D FE A7
28 F6 E1 77 23 E5 01 FF FF ED 42 20 ED E1
FD E1 DD E1 C9 DD 21 CB FD 11 FF 00 DD
19 FD 7E 00 FD AE 01 DD A6 01 FD AE 01
43 04 0F 10 FD 17 1D 1D DD 21 FF FF 38 12
17 38 05 DD 7E 20 18 2F 1D 17 DD 7E 41 30
28 AF 18 25 1F 1D 1D 1D FE F8 30 18 C6 F8
CB 2F F6 80 FE F4 30 0F C6 F4 CB 2F FE EA
30 08 C6 EA CB 2F 18 03 1D 1D 1D C6 20 21
69 FD 4F 09 7B FE 80 38 04 C6 08 FD 23 32
72 FE DD 7E 00 FE 5F 7E 20 0B DD 21 89 FD
DD 22 8C FE D6 20 C9 FE 5E 20 07 CD 6D FE
20 02 C6 20 22 8C FE FE 5F CA 6D FE FE 20
C8 EE 20 C9
```

LOAD your existing source code from tape, if you have decided to use the assembly listing. Add the code that follows, and assemble it according to the instructions given with your assembler. Make sure that you have a tape to hand to SAVE the assembled code.

```
JSR  $5341
BYT  $4F
BYT  $54
BYT  $52
EOR  #$4C
EOR  $43
LSR  $555F,X
EOR  $5750
EOR  $424E,Y
BYT  $47
BYT  $44
LSR  $56
PHA
BYT  $4B
EOR  ($58),Y
LSR  A
BYT  $5A
BYT  $5B
BYT  $5C
EOR  $110A,X
BYT  $04
ORA  ($17),Y
PHP
ORA  ($08),Y
ORA  $17
BYT  $02
BYT  $07
ORA  ($08),Y
BYT  $03
ORA  ($01,X)
BYT  $14
PHP
PHP
PHP
BYT  $04
PHP
PHP
PHP
BYT  $0C
ASL  $080C
BYT  $0B
BYT  $0B
BRK
BYT  $0B
BYT  $04
BYT  $04
PHP
ORA  $06,X
BYT  $03
BYT  $04
ASL  $14
BYT  $03
BYT  $04
ORA  ($02),Y
ORA  ($08,X)
BYT  $17
BYT  $03
BYT  $04
BYT  $07
BYT  $17
ASL  $03
ASL  $01
ASL  $1A
BYT  $04
BYT  $03
ASL  $0A
ASL  $0A0B,X
BRK
```





```
ORA  ($Ø3,X)
BYT  $Ø7
BYT  $ØF
BYT  $1F
BYT  $3F
BYT  $7F
BYT  $FF
INC  $F8FC,X
BEQ  $CECD
CPY  #$8Ø
BRK
PHA
TYA
PHA
LDY  #$ØØ
LDA  ($2D),Y
INY
CMP  #$5A
BNE  $CFØ2
LDA  ($2D),Y
CMP  #$8Ø
BEQ  $CFØ8
TYA
ADC  #$Ø6
TAY
BNE  $CEF5
INY
STY  $CF48

PLA
TAY
PLA
RTS
PHA
TXA
PHA
TYA
PHA
LDA  $2F
STA  $61
LDA  $3Ø
STA  $62
LDY  #$ØØ
LDA  ($61),Y
CMP  #$DA
BNE  $CF2C
INY
LDA  ($61),Y
CMP  #$8Ø
BEQ  #CF34
LDY  #$Ø2
CLC
JSR  $CE6B
BCC  $CF1D
LDY  #$Ø7
SEC
JSR  $CE6B

LDA  #$Ø7
STA  $CF84
LDA  #$ØØ
STA  $CF9Ø
STA  $CF5C
LDY  #$FF
LDA  ($2D),Y
STA  $CF5E
INY
LDA  ($2D),Y
STA  $65
INY
LDA  ($2D),Y
STA  $66
JSR  $CF72
LDY  #$FF
CPY  #$FF
BEQ  $CF6C
STA  ($65),Y
INC  $CF5C
TAX
BNE  $CF58
INY
BNE  $CF5D
PLA
TAY
PLA
TAX

PLA
RTS
LDY  #$ØØ
LDX  $CF84
LDA  ($61),Y
INY
ROR  A
EOR  ($61),Y
AND  $CEDF,X
EOR  ($61),Y
CLV
BVC  $CF85
ROL  A
ROL  A
ROL  A
ROL  A
ROL  A
ROL  A
ROL  A
ROL  A
DEX
DEX
LDY  #$FF
ROL  A
BCS  $CFA8
BMI  $CF9B
LDA  $CE9D,Y
BCC  $CFC9

DEX
ROL  A
BMI  $CFA4
LDA  $CEBE,Y
BCS  $CFC9
LDA  #$ØØ
BEQ  $CFC9
ROR  A
DEX
DEX
DEX
CMP  #$F8
BPL  $CFC4
ADC  #$F8
ROR  A
CMP  #$F4
BPL  $CFC5
ADC  #$F4
ROR  A
CMP  #$EA
BPL  $CFC6
ADC  #$EA
ROR  A
SEC
BMI  $CFC7
DEX
DEX
DEX

ADC  #$1F
PHA
TXA
BPL  $CFD5
EOR  #$F8
INC  $61
BNE  $CFD5
INC  $62
STA  $CF84
PLA
TAX
LDA  $CE7D,X
CPY  #$ØB
BEQ  $CFF3
CMP  #$5E
BNE  $CFEB
JSR  $CF72
CLC
ADC  #$2Ø
STX  $CF9Ø
CMP  #$5F
BEQ  $CF72
RTS
SEC
SBC  #$2Ø
LDY  #$2Ø
STY  $CF9Ø
RTS
```

If you don't have an assembler you will need to enter the hexadecimal listing that follows instead of the assembly language listing. You will need a hex loading program to enter the code, though, which varies slightly according to whether you have a Vic 20 or a Commodore 64.

If you have a Commodore 64, type this line first:

```
5 POKE 53280,1:POKE 53281,1
```

Vic 20 owners should use instead:

```
5 POKE 36879,25
```

The remainder of the program is common to both machines.

Type in these lines and then you can RUN the program:

```
8 HH$="0123456789ABCDEF"
10 PRINT "■■■■";TAB(6);"HEX
   MONITOR"
15 PRINT "■■■■■1)□ENTER
   MACHINE CODE"
30 PRINT "■■2)□SAVE BYTES"
35 PRINT "■■■■■■■ENTER
   NUMBER (1-2)?■"
40 GET A$:IF A$<"1" OR A$>"2" THEN
   40
50 ON VAL(A$) GOSUB 100,300
60 GOTO 10
100 SA$="":INPUT"■START
   ADDRESS□";SA$:PRINT "■"
102 D$=LEFT$(SA$,2):GOSUB
   400:S1=C:IF W<2 THEN 100
104 D$=RIGHT$(SA$,2):GOSUB 400:IF
   W<2 THEN 100
105 SA=S1*256+C
110 GOSUB 450:DD$="":INPUT DD$:IF
   DD$="" THEN 10
120 ZZ=0:D$="":DD$=DD$+"□":
   FOR LL=1 TO LEN(DD$):IF MID$
   (DD$,LL,1)="□" THEN 130
122 D$=D$+MID$(DD$,LL,1):GOTO 160
130 ZZ=ZZ+1:GOSUB 400:IF W<2 THEN
   PRINT:GOTO 110
150 POKE SA,C:D$="":SA=SA+1
160 NEXT LL:GOTO 110
300 CLR:HH$="0123456789ABCDEF":
   SA$="":INPUT"■■START
   ADDRESS■";SA$
302 D$=LEFT$(SA$,2):GOSUB
   400:AA=C:IF W<2 THEN 300
304 D$=RIGHT$(SA$,2):GOSUB
   400:A2=C:IF W<2 THEN 300
310 INPUT "■□END ADDRESS□■";SA$
312 D$=LEFT$(SA$,2):GOSUB 400:BB=C:
   IF W<2 THEN 310
314 D$=RIGHT$(SA$,2):GOSUB 400:
   B2=C:IF W<2 THEN 310
315 INPUT "■□□FILE NAME□□■";
   N$:IF N$="" THEN 315
316 D=1:INPUT "■(D)ISK,(T)APE■";
   D$:IF D$="D" THEN
   N$="@:"+N$:D=8
320 PRINT "■P┌44,";AA;
   ":P┌43,";A2
330 PRINT "■■P┌46,";BB;
   ":P┌45,";B2
335 PRINT "■■SAVE";CHR$(34);
   N$;CHR$(34);",";D;",1"
340 PRINT "■■■■■■■■■
   P┌44,";PEEK(44);":P┌43,";
   PEEK(43)
350 PRINT "■■P┌46,";PEEK(46);
   ":P┌45,";PEEK(45):PRINT
   "■■RUN■"
399 END
400 W=0:FOR Z=1 TO 16:IF LEFT$(D$,1)
   =MID$(HH$,Z,1) THEN W=W+1
410 IF RIGHT$(D$,1)=MID$(HH$,Z,1) THEN
   W=W+1
420 NEXT:IF W<2 THEN RETURN
430 A=ASC(D$)-48:B=ASC(RIGHT$
   (D$,1))-48
440 C=B+7*(B>9)-(LEN(D$)=2)*(16*
   (A+7*(A>9))):RETURN
450 S1=INT(SA/256):S2=SA-S1*256:
   S(1)=INT(S1/16):S(2)=S1-S(1)*16
453 S(3)=INT(S2/16):S(4)=S2-
   S(3)*16
455 FOR L=1 TO 4
460 PRINT MID$(HH$,S(L)+1,1);
480 NEXT L:PRINT "□";:RETURN
```

The start addresses and the alterations you'll need to make if you don't have a Commodore 64, along with the POKEs needed for some memory sizes in the Vic are on page 632.

If you have a Vic, you'll need to change the numbers in bold type in the hex listing—to find out how to do this, you should also turn to page 632.

A2 1F 48 38 FD 7D **CE** F0 22 C9 80 F0 04 C9
20 D0 0B 8A 48 A9 5E 20 14 **CD** 68 AA 10
0F C9 E0 D0 06 8A 48 A9 5F D0 EF 68 CA 10
D8 60 68 8A A0 FF C9 0A F0 03 8D 42 **CD**
AE 99 **CD** CA CA D9 9D **CE** D0 04 A9 00 F0
31 CA D9 BE **CE** D0 04 A9 40 D0 27 C9 01
10 04 A9 60 10 1F CA CA CA CA CA 69 DF
C9 F8 10 14 E8 0A 69 07 C9 F0 10 0C E8 0A
69 0F C9 C8 10 04 E8 0A 69 37 C0 0B D0 05
A0 20 8C 42 **CD** 8E B5 **CD** AE 99 **CD** D0 00
6A 6A 6A 6A 6A 6A 6A 6A 48 2A 3D E0 **CE**
A0 00 11 61 91 61 C8 68 3D E8 **CE** 91 61 A9
FF 10 08 49 F8 E6 61 D0 02 E6 62 8D 99 **CD**
A9 00 60 48 98 48 A0 08 8C 1A **CE** A0
00 8C 13 **CE** B1 2D C8 C9 5A D0 06 B1 2D
C9 80 F0 06 98 69 06 A8 D0 ED C8 8C 2F **CE**
68 A8 68 60 48 8A 48 98 48 A5 2F 85 61 A5
30 85 62 A0 00 B1 61 C9 DA D0 07 C8 B1
61 C9 80 F0 08 A0 02 18 20 6B **CE** 90 E9 A9
FF A0 08 91 61 88 A9 FF 91 61 38 20 6B **CE**
A9 07 8D 99 **CD** A9 00 8D 42 **CD** 8D 40 **CE**
A0 FF B1 2D 8D 42 **CE** C8 B1 2D 85 65 C8
B1 2D 85 66 A0 FF C0 FF F0 0A B1 65 20 14
CD EE 40 **CE** D0 F0 20 14 **CD** A9 06 **CD** 99
CD A5 61 E5 63 8D 1A **CE** A5 62 E5 64 8D 13
CE 68 A8 68 AA 68 60 A5 61 85 63 71 61 85
61 C8 A5 62 85 64 71 63 85 62 60 20 41 53
4F 54 52 49 4C 45 43 5E 5F 55 4D 50 57 59
4E 42 47 44 46 56 48 4B 51 58 4A 5A 5B 5C
5D 0A 11 04 11 17 08 11 08 05 17 02 07 11
08 03 01 01 14 08 08 08 04 08 08 08 0C 0E
0C 08 0B 0B 00 0B 04 04 08 15 06 03 04
06 14 03 04 11 02 01 08 17 03 04 07 17 06
03 06 01 06 1A 04 03 06 0A 1E 0B 0A 00
01 03 07 0F 1F 3F 7F FF FE FC F8 F0 E0 C0
80 00 48 98 48 A0 00 B1 2D C8 C9 5A D0
06 B1 2D C9 80 F0 06 98 69 06 A8 D0 ED
C8 8C 48 **CF** 68 A8 68 60 48 8A 48 98 48 A5
2F 85 61 A5 30 85 62 A0 00 B1 61 C9 DA D0
07 C8 B1 61 C9 80 F0 08 A0 02 18 20 6B **CE**
90 E9 A0 07 38 20 6B **CE** A9 07 8D 84 **CF**
A9 00 8D 90 **CF** 8D 5C **CF** A0 FF B1 2D 8D
5E **CF** C8 B1 2D 85 65 C8 B1 2D 85 66 20 72
CF A0 FF C0 FF F0 0B 91 65 EE 5C **CF** AA
D0 EF C8 D0 F1 68 A8 68 AA 68 60 A0 00
AE 84 **CF** B1 61 C8 6A 51 61 3D DF CE 51 61
B8 50 00 2A 2A 2A 2A 2A 2A 2A 2A CA CA
A0 FF 2A B0 14 30 05 B9 9D **CE** 90 2E CA
2A 30 05 B9 BE **CE** B0 25 A9 00 F0 21 6A
CA CA CA C9 F8 10 14 69 F8 6A C9 F4 10 0E
69 F4 6A C9 EA 10 08 69 EA 6A 38 30 03 CA
CA CA 69 1F 48 8A 10 08 49 F8 E6 61 D0 02
E6 62 8D 84 CF 68 AA BD 7D **CE** C0 0B F0
12 C9 5E D0 06 20 72 **CF** 18 69 20 8E 90 **CF**
C9 5F F0 80 60 38 E9 20 A0 20 8C 90 **CF** 60
00 0A

LOAD the existing source code off tape ready for the remainder of the assembly listing—the decoder. Once you have the completed listing in the machine, SAVE the source code then assemble it by typing RUN. Make sure that you have a tape to hand, ready for when the machine prompts you to press the record key, to SAVE the code on tape.

```
1200 FOR T=0 TO 3 STEP 3
1210 P%=ESETUP
1220 [OPT T
1230 .DECODE□LDY #0
1240 LDX BRAN+1: LDA (ZCOD),Y
1250 INY: ROR A: EOR (ZCOD),Y
1270 AND LO,X: EOR (ZCOD),Y
1280 CLV: .BRAN□BVC BRAN+2
1290 ROL A: ROL A: ROL A
1300 ROL A: ROL A: ROL A
1310 ROL A: ROL A: DEX: DEX
1320 .CHAR□LDY #&FF: ROL A
```

```
1330 BCS LONGD: BMI THREE
1340 LDA FIRST,Y: BCC FOUND
1350 .THREE□DEX: ROL A
1360 BMI SPACE: LDA SECOND,Y
1370 BCS FOUND
1380 .SPACE□LDA #Ø
1390 BEQ FOUND
1400 .LONGD□ROR A: DEX: DEX
1410 DEX: CMP #255+
    CEIGHT-CLAST
1420 BPL EIG: ADC #255+
    CEIGHT-CLAST
1430 ROR A: CMP #255+
    CSEVEN-CLAST
1440 BPL SEV: ADC #255+
    CSEVEN-CLAST
1450 ROR A:CMP #255+
    CSIX-CLAST
1460 BPL SIX: ADC #255+
    CSIX-CLAST
1470 ROR A: SEC:
1480 BMI FIV: .EIG□DEX
1490 .SEV□DEX: .SIX□DEX
1500 .FIV□ADC #CLAST-CODE
1510 .FOUND□PHA: TXA
1520 BPL NOCHANGE: EOR #&F8
1530 INC ZCOD: BNE NOCHANGE
1540 INC ZCOD+1
1550 .NOCHANGE□STA BRAN+1
1560 .FETCH□PLA: TAX
1570 LDA CODE,X
1580 CPY #CAR-CODE
1590 BEQ LOWER
1600 .JB□BEQ DECODE
1610 CMP #&5E
1620 BNE NTUPPER: JSR DECODE
1630 BNE NTUPPER: LDA #&4Ø
1640 .NTUPPER□STX CHAR+1
1650 CMP #&5F: BEQ JB
1660 CMP #&2Ø: BEQ RTS
1670 EOR #&2Ø:.RTS□RTS
1680 .LOWER□SEC: SBC #&2Ø
1690 LDY #CPUN-CODE
1700 STY CHAR+1: RTS
1710 .DSTRING□LDY #4
1720 LDA (ZBOX),Y:STA ZDOL
1730 INY: LDA (ZBOX),Y
1740 STA ZDOL+1: LDY #7
1750 LDA (ZBOX),Y
1760 STA LCNZD+1:STY BRAN+1
1770 INY: LDA (ZPCT),Y: INY
1780 CLC: ADC ZPCT: STA ZCOD
1790 LDA (ZPCT),Y
1800 ADC ZPCT+1:STA ZCOD+1
1810 LDA #Ø: STA CHAR+1
1820 STA MSTER+1
1830 .MD□JSR DECODE
1840 .MSTER□LDY #&FF
1850 .LCNZD□CPY #&FF
1860 BEQ XIT: STA (ZDOL),Y
1870 INC MSTER+1: TAX: BNE MD
1880 INY: BNE LCNZD: .XIT□RTS
1890 ]:NEXT
1900 *SAVE"DECODE" 5EØØ 5F8Ø
1910 END
1920 DATA ØA11Ø41117Ø811Ø8Ø517Ø2
    Ø711Ø8Ø3Ø1Ø114Ø8Ø8Ø8Ø4Ø8Ø8Ø8ØC
    ØEØCØ8ØBØBØØØB
1930 DATA Ø4Ø4Ø815Ø6Ø3Ø4Ø614
    Ø3Ø411Ø2Ø1Ø817Ø3Ø4Ø717Ø6Ø3Ø6
    Ø1Ø61AØ4Ø3Ø6ØA1EØBØAØØØ1Ø3
    Ø7ØF1F3F7FFFFEFCF8FØEØCØ
    8ØØØ
```

The assembly listing below is for the decoder. It should be assembled starting at memory location 3238Ø. When you are ready to SAVE it, remember to use a separate tape from the coder, which you finished last time.

```
@CODE EQU *
@CSP FCB $2Ø
@CA  FCB !A
@CS  FCB !S
@CO  FCB !O
@CT  FCB !T
@CR  FCB !R
@CI  FCB !I
@CL  FCB !L
@CE  FCB !E
     FCB !C
@CSIX EQU *
@CUP FCB !^
@CAR FCB $5F
@CU  FCB !U
     FCB !M
@CP  FCB !P
     FCB !W
     FCB !Y
@CN  FCB !N
     FCB !B
     FCB !G
@CSEVEN EQU *
@CD  FCB !D
@CF  FCB !F
     FCB !V
@CH  FCB !H
@CEIGHT EQU *
     FCB !K
     FCB !Q
@CX  FCB !X
     FCB !J
     FCB !Z
     FCB $5B
@CPO FCB $5C
@CLAST FCB $5D
@CPUN EQU *
@FIRST FCB @CUP-@CODE
     FCB @CN-@CODE
     FCB @CT-@CODE
     FCB @CN-@CODE
     FCB @CH-@CODE
     FCB @CE-@CODE
     FCB @CN-@CODE
     FCB @CE-@CODE
     FCB @CR-@CODE
     FCB @CH-@CODE
     FCB @CS-@CODE
     FCB @CL-@CODE
     FCB @CN-@CODE
     FCB @CE-@CODE
     FCB @CO-@CODE
     FCB @CA-@CODE
     FCB @CA-@CODE
     FCB @CD-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CT-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CU-@CODE
     FCB @CP-@CODE
     FCB @CU-@CODE
     FCB @CE-@CODE
     FCB @CAR-@CODE
     FCB @CAR-@CODE
     FCB @CSP-@CODE
     FCB @CAR-@CODE
@SECOND FCB @CT-@CODE
     FCB @CT-@CODE
     FCB @CE-@CODE
     FCB @CF-@CODE
     FCB @CI-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CI-@CODE
     FCB @CD-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CN-@CODE
     FCB @CS-@CODE
     FCB @CA-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CH-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CL-@CODE
     FCB @CH-@CODE
     FCB @CI-@CODE
     FCB @CO-@CODE
     FCB @CI-@CODE
     FCB @CA-@CODE
     FCB @CI-@CODE
     FCB @CX-@CODE
     FCB @CT-@CODE
     FCB @CO-@CODE
     FCB @CI-@CODE
     FCB @CUP-@CODE
     FCB @CPO-@CODE
```

```
        FCB @CAR—@CODE
        FCB @CUP—@CODE
@LO     FCB Ø
        FCB $1
        FCB $3
        FCB $7
        FCB $F
        FCB $1F
        FCB $3F
        FCB $7F
@UP     FCB $FF
        FCB $FE
        FCB $FC
        FCB $F8
        FCB $FØ
        FCB $EØ
        FCB $CØ
        FCB $8Ø
        FCB Ø
@USR7 PSHS D
        JSR $8B3Ø
        STD @ZPTR2+1
        PULS D
        RTS
@USR6 PSHS A,B,X,Y,U
        LDA #7
        STA @BITTY+2
        LDD #@CSP
        STD @CHAR+1
        JSR $8B3Ø
        TFR D,Y
@ZPTR2 LDU #$ABCD
        PULU B
        PULU A
        ADDD ,U
        LDU ,U
        PSHS D
        LDA #$F
        BRA @DCAMP
@DCLOOP CMPA #Ø
        BEQ @DCSTR
        PSHS U
        JSR @DECODE
        PULS U
@DCSTR STA ,U+
@DCAMP CMPU ,S
        BNE @DCLOOP
@DCEX PULS D
        PULS A,B,X,Y,U
        RTS
@DECODE EQU *
@BITTY LDX #$FF
        TFR X,D
        EXG A,B
        LDB ,Y
        LSRB
        EORB 1,Y
        ANDB @LO,X
        EORB 1,Y
        STA @BRAN+1
@BRAN BRA□*+$F
        ROLB
        ROLB
        ROLB
        ROLB
        ROLB
        ROLB
        ROLB
        ROLB
@CHAR LDX #$ABCD
        ROLB
        BCS @LONGD
        BMI @THREE
        LDB @FIRST—@CODE,X
        BRA @FOUND
@THREE DECA
        ROLB
        BMI @SPACE
        LDB @SECOND—@CODE,X
        BRA @FOUND
@SPACE LDB #Ø
        BRA @FOUND
@LONGD RORB
        SUBA #3
        CMPB #@CEIGHT—@CLAST+255
        BPL @EIG
        ADDB #@CEIGHT—@CLAST—1
        RORB
        CMPB #@CSEVEN—@CLAST—1
        BPL @SEV
        ADDB #@CSEVEN—@CLAST—1
        RORB
        CMPB #@CSIX—@CLAST—1
        BPL @SIX
        ADDB #@CSIX—@CLAST—1
        RORB
        BRA @FIV
@EIG DECA
@SEV DECA
@SIX DECA
@FIV ADDB #@CLAST—@CODE+1
@FOUND SUBA #2
        BPL@NOCHNGM
        ADDA #8
        LEAY 1,Y
        @NOCHNG STA @BITTY+2
        @FETCH CLRA
        ADDD #@CODE
        TFR D,U
        LDA ,U
        CMPX #@CAR
        BEQ @LOWER
        CMPA #!∧
        BNE @NOTUPP
        JSR @DECODE
        ADDA #$2Ø
@NOTUPP STU @CHAR+1
        CMPA #$5F
        LBEQ @DECODE
        RTS
@LOWER SUBA #$2Ø
        LDU #@CLAST+1
        BRA @NOTUPP
        END
```

If you don't have a commercial assembler for your Dragon or Tandy, you'll need this BASIC program so that you can enter the hex code into your machine:

```
10 CLS:
15 CLEAR 255, 32379
20 INPUT "START□ADDRESS□";SA
30 LINE INPUT D$
40 IF D$="" THEN GOTO 30
50 IF LEFT$(D$,1)="#" THEN END
60 S$=LEFT$(D$,2)
70 POKE SA,VAL("&H"+S$)
80 PRINTSA,LEFT$(D$,2)
90 D$=MID$(D$,3)
100 SA=SA+1:GOTO 40
```

To enter the hex listing, you can either do so one number at a time, or you can enter a block of any size *without* spaces, up to the maximum INPUT size. Then, to end the program, you should enter a #.

RUN the program, and enter the start address—3238Ø. You can now enter the hex code for the encoder. Tandy owners should note that they will have to change the two 8B 3Øs to B3 ED, and the 8C 37 to B4 F4.

```
2Ø 41 53 4F 54 52 49 4C 45 43 5E 5F 55 4D
5Ø 57 59 4E 42 47 44 46 56 48 4B 51 58 4A
5A 5B 5C 5D ØA 11 Ø4 11 17 Ø8 11 Ø8 Ø5 17
Ø2 Ø7 11 Ø8 Ø3 Ø1 Ø1 14 Ø8 Ø8 Ø8 Ø4 Ø8 Ø8
Ø8 ØC ØE ØC Ø8 ØB ØB ØØ ØB Ø4 Ø4 Ø8 15
Ø6 Ø3 Ø4 Ø6 14 Ø3 Ø4 11 Ø2 Ø1 Ø8 Ø8 17 Ø3
Ø4 Ø7 17 Ø6 Ø3 Ø6 Ø1 Ø6 1A Ø4 Ø3 Ø6 ØA 1E
ØB ØA ØØ Ø1 Ø3 Ø7 ØF 1F 3F 7F FF FE FC F8
FØ EØ CØ 8Ø ØØ 34 Ø6 BD 8B 30 FD 7F 1E 7F
7F 16 7F 7F 17 35 Ø6 39 34 76 86 Ø7 B7 7F
D7 CC 7E 7C FD 7F 81 BD 8B 30 1F Ø1 34 1Ø
CC AB CD 31 8B BD 8C 37 CE AB CD 37 Ø4
37 Ø2 E3 C4 EE C4 FD 7F 38 2Ø Ø9 37 Ø2 34
4Ø BD 7F 55 35 4Ø 11 83 AB CD 26 F1 4F BD
7F 55 81 Ø7 27 Ø2 A6 AØ 35 Ø6 43 53 C3 ØØ
Ø1 3Ø AB BF 7F 16 35 76 39 CE 7E 9C 1F 89
AØ C2 27 2Ø 81 2Ø 26 ØB 86 5E 34 4Ø BD 7F
55 35 4Ø 2Ø 11 81 EØ 26 Ø4 86 5F 2Ø EF 1F
98 11 83 7E 7B 2E DD 39 1F 3Ø 8E AB CD B6
7F D7 B7 7F D4 8Ø Ø2 CØ 86 27 ØB 8C 7E 87
26 Ø3 CE 7E 9C FF 7F 81 CB ØA 27 13 E1 88
2Ø 26 Ø4 C6 ØØ 2Ø 2B 4A E1 88 41 26 Ø9 C6
4Ø 2Ø 21 C6 6Ø 4A 2Ø 1C 8Ø Ø5 CØ 2Ø C1 F8
2A 14 4C 58 CB Ø8 C1 FØ 2A ØC 4C 58 CB
1Ø C1 C8 2A Ø4 4C 58 CB 38 8E ØØ FF 54 2Ø
ØD 56 56 56 56 56 56 56 34 Ø4 59 E4 89 7E
EØ EA A4 E7 A4 35 Ø4 E4 89 7E E8 E7 21 81
ØØ 2C Ø4 8B Ø8 E6 AØ B7 7F D7 39
```

Now SAVE the machine code by typing CSAVEM"CODER", 3238Ø, 32766, 3238Ø and then press RETURN.

The decode part of the text compressor will be stored separately from the encode half. LOAD the hex loading program and RUN it. The start address is the same as last time—32380—and if you have a Tandy, the 8B 30s will have to be changed to B3 ED:

```
20 41 53 4F 54 52 49 4C 45 43 5E 5F 55 4D
50 57 59 4E 42 47 44 46 56 48 4B 51 58 4A
5A 5B 5C 5D 0A 11 04 11 17 08 11 08 05 17
02 07 11 08 03 01 01 14 08 08 08 04 08 08
08 0C 0E 0C 08 0B 0B 00 0B 04 04 08 15
06 03 04 06 14 03 04 11 02 01 08 08 17 03
04 07 17 06 03 06 01 06 1A 04 03 06 0A 1E
0B 0A 00 01 03 07 0F 1F 3F 7F FF FE FC F8
F0 E0 C0 80 00 34 06 BD 8B 30 FD 7F 0E
35 06 39 34 76 86 07 B7 7F 37 CC 7E 7C FD
7F 55 BD 8B 30 1F 02 CE AB CD 37 04 37 02
E3 C4 EE C4 34 06 86 0F 20 0D 81 00 27 07
34 40 BD 7F 35 35 40 A7 C0 11 A3 E4 26 EE
35 06 35 76 39 8E 00 FF 1F 10 1E 89 E6 A4
54 E8 21 E4 89 7E DF E8 21 B7 7F 4B 20 0D
59 59 59 59 59 59 59 59 8E AB CD 59 25 14
2B 05 E6 88 20 20 2C 4A 59 2B 05 E6 88 41
20 23 C6 00 20 1F 56 80 03 C1 F8 2A 13 CB
F8 56 C1 F4 2A 0D CB F4 56 C1 EA 2A 07 CB
EA 56 20 03 4A 4A 4A CB 20 80 02 2A 04 8B
08 31 21 B7 7F 37 4F C3 7E 7C 1F 03 A6 C4
8C 7E 87 27 13 81 5E 26 05 BD 7F 35 8B 20
FF 7F 55 81 5F 10 27 FF 7E 39 80 20 CE 7E
9C 20 EF
```

Now SAVE the machine code by typing CSAVEM"DECODER", 32380, 32703, 32380 and press RETURN.

USING THE COMPRESSOR

Now that you have both the encoder and decoder on tape you can put the compressor to use. Next time you can see how to use it to develop an adventure game.



DETECTING THINGS ON SCREEN

Commands which let your computer 'look at' its own screen allow it to keep track of complicated graphics. This can be especially useful for games where collisions occur

If you have already created a detailed screen display, how do you ensure that the next object you put into the picture does not coincide with something that's been put there earlier?

One answer is to be careful and systematic about keeping track of which parts of the screen have been used. But this can get pretty complicated, and there are circumstances where it is next to impossible—if you have moving graphics, for example. Yet this is precisely the sort of problem you encounter in writing things like games programs, where you might have aliens and spaceships moving around a galaxy—or ghosts moving around a maze. In either case, you want to ensure that two graphics do not get printed in the same space, or that particular things happen if they collide—the spaceship blows up, say.

DETECTING STRANGE SHAPES

For example, suppose you want to write a program to bounce a ball around the screen. This is quite simple: you know the coordinates of the four sides of the screen, and so you can just include four IF ... THEN conditions to check whether or not the ball is hitting the side of the screen. This is done by comparing the ball's coordinates with the known coordinates of the sides. But what happens if you want to check to see whether a ball has hit the sides of a strangely shaped object—a circle, say?

You could use the same method: have a number of IF ... THEN checks which contain details of the circle's coordinates to see when the ball hits the sides. But since the curved sides are a relatively complex shape, you would need a very large number of checks. And as you know, the computer takes a long time to execute each IF ... THEN check. A program written in this way would become very, very slow, and the movement on the screen would appear jerky.

There are limits to the speed you can obtain using BASIC. But the Spectrum, Commodore, Acorn, Dragon and Tandy have a command which makes it possible to detect the presence of any object on screen more quickly than checking for coordinates. This is possible even if you don't know its position.

COLOUR BY NUMBERS

The Spectrum command, ATTR, the Commodores', PEEK, the Acorn, Dragon and Tandy command, POINT or PPOINT, all return as an answer, the colour(s) in each specified character square (or pixel position, with the Acorn's POINT and the Dragon and Tandy's PPOINT).

This means you can detect the presence of any object, simply by specifying its colour and then checking all the screen positions. So in the earlier example, a red circle would return the number code for red wherever it appeared. Simply by checking for this code, the program could then make the ball bounce off the circle, or whatever.

The Spectrum actually gives a number between Ø and 225, which takes into account all the ATTRibutes of each character square: the PAPER and INK colours, whether BRIGHT is on, and whether or not the character is FLASHing. How you arrive at the number of each character square's ATTRibutes is explained on pages 68 and 69.

The Commodore 64 and Vic effectively give an answer between Ø and 15—the colour number. In fact the Commodore 64 adds 24Ø to each number, and so gives you a number between 24Ø and 255. All you do is subtract 24Ø from the number you are given, to leave the number of the colour. So, for example, if you PEEKed a certain character's colour address, and got the response 24Ø, your answer is Ø, or a black character.

The Acorn computers give a number between −1 and 15. −1 is given when you are asking for the colour of a point which is outside the limits of the screen. The numbers between Ø and 15 refer to the logical colour of the relevant pixel.

The Dragon and Tandy simply give a number between −1 and 8 which refers to the colour of the character square, or, for PPOINT, the pixel position. The curious result −1 occurs when you try to find out the colour of a letter: text can only be either green or black (and then the black colour is actually just the INVERSE of green).

The syntax of each of these commands is much the same, except on the Commodore. The command, ATTR or whatever, must be followed by the screen coordinates of the character position whose colour you want to know, in brackets. So you might use this:

```
PRINT ATTR(1Ø,1Ø)
```

The Commodore is slightly different, in that instead of using screen coordinates in brackets to tell the computer which square you are interested in, you PEEK an actual memory address which holds the details of part of the screen display. The address of the first character square is 55296, and the last address is 56295. The 'file' takes up exactly 1ØØØ bytes—the screen is made up from 25 lines of 40 characters.

The first address refers to the character square at the top left-hand corner of the screen, the next address to the next character in the line, and so on. This means that the bottom right-hand corner character's colour details are held by address 56295. The command thus takes the form:

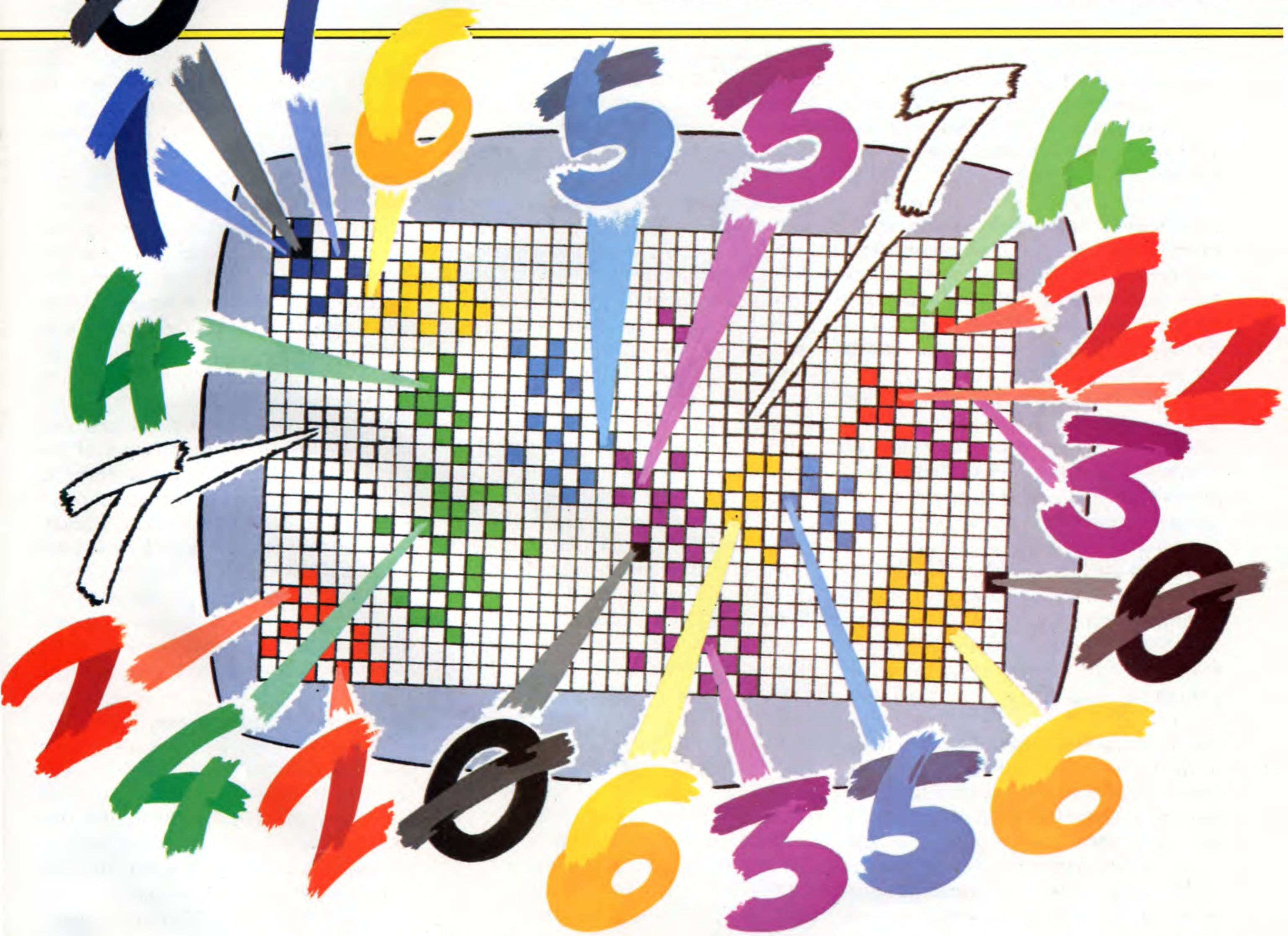
```
PRINT PEEK (56296) AND 15
```

(The last part of this returns a number from Ø to 15). Unfortunately, you can't reverse the process and change the colours of each character square. You can, however, POKE different values into the attribute file on the Spectrum and Commodore (the Dragon, Tandy and Acorn do not have a separate colour file, but change colour by PRINTing different coloured characters).

Type in and RUN the following program which is an example of where you might want to use the POINT, ATTR, PEEK commands.

```
1Ø BORDER Ø: PAPER Ø: INK 9
15 CLS
2Ø FOR n=22528 TO 22559: POKE n,48:
   POKE n+672,48: NEXT n
3Ø FOR n=2256Ø TO 23168 STEP 32: POKE
   n,48: POKE n+31,48: NEXT n
4Ø FOR n=1 TO 3Ø: PRINT PAPER 6;AT INT
   (RND*8)*2+3,INT (RND*13)*2+3;"□":
   NEXT n
5Ø LET x=15: LET y=1Ø: LET xv=−1: LET
   yv=1
8Ø PRINT AT y,x;"O": LET oy=y:
   LET ox=x
```

- KEEPING TRACK OF SCREEN DISPLAYS
- DETECTING STRANGE SHAPES
- COLOUR CHECKS
- **ATTR, PEEK** AND **POINT**
- BOUNCING BALL PROGRAM
- COLLISION DETECTION
- NEW DIRECTIONS



```
90 LET x=x+xv: LET y=y+yv
140 IF ATTR (y,x−1)=48 OR ATTR
    (y,x+1)=48 THEN LET xv=−xv
145 IF ATTR (y−1,x)=48 OR ATTR
    (y+1,x)=48 THEN LET yv=−yv
190 PRINT AT oy,ox;"□": GOTO 80
```

The Spectrum's display consists of a series of randomly PRINTed yellow spaces inside a yellow border. When you RUN the program, the computer starts to move a ball in an initially random direction.

Whenever the ball hits either the border or one of the blocks, it bounces away in a different direction.

DETECTING COLLISIONS

The checks to see when the ball hits the border or one of the squares are done with the command ATTR. Since the squares are PRINTed randomly, the only alternative would be to keep track of them with a variable for every x and every y coordinate. If you did use two variables for each square, quite apart from using up a large amount of memory, you would need a huge amount of IF ... THEN checks. These would make what is now a reasonably fast program into an impossibly slow one.

You easily could use IF ... THEN checks to detect when the ball is hitting the border, since its coordinates are easily defined. But once you have used ATTR in a check to see when the ball is hitting a yellow block, it makes sense to use it for the border as well—especially where this is the same colour as the blocks, which it is for the program above.

The program starts off by setting up the display colours, and creating the yellow frame. Program Lines 20 and 30 which add the frame are interesting in that instead of PRINTing spaces or blank ROM graphics, the frame is set up by POKEing into the attribute file. Two FOR ... NEXT loops step through the memory addresses which hold the attributes

for the character squares around the edge of the screen. Each address is POKEd with 48, the code for yellow PAPER and black INK (with BRIGHT and FLASH switched off).

The program continues by PRINTing random blocks (spaces, this time). The random numbers which control the PRINT positions are designed to work within an area two characters squares away from the edge of the screen (to prevent any of the random squares being wasted by falling on the frame).

Line 5Ø initializes a few variables, x and y for the initial position of the ball, and xv and yv for the starting direction of the ball in the x and the y (the horizontal and the vertical) axes.

Once this is done, the computer jumps to Line 8Ø where it PRINTs the ball and sets up another two variables: ox and oy to hold the old values of x and y. These are used to rub out the ball once x and y have been changed by the subsequent calculations which determine the movement of the ball.

The calculations are done in Line 9Ø. They simply add the direction vectors (or the distance travelled in each direction) which are determined by xv and xy to both the x and y coordinate.

USING ATTR

Lines 14Ø and 145 are the important lines which check the colours of the four squares around the ball.

As you can see, the ATTR function looks like this:

ATTR (y coordinate, x coordinate)

But like PEEK, ATTR is not a direct command. You cannot, then, tell the computer to ATTR as you might tell it to PRINT or LOAD. It is thus contained within a command—in this case an IF . . . THEN statement.

The LET statement after the conditions in both of these lines simply reverses the speed vector IF the colour of the next square that the ball would move onto is yellow (ATTR = 48 is a yellow character square).

What this does in practice is to make the ball bounce away at an angle from the square it has just 'hit'. The ball must be coming from either above or below (in the y direction) *or* beside the obstacle (in the x direction) at the moment it hits. If both speed vectors were reversed, the ball would simply back-track itself. So the checks only affect its speed in the direction in which it hits. Its velocity in the other direction (the direction in which it has *not* hit) remains the same. So, if it is detected to be about to hit a yellow block below it, it will start to move upwards, but its horizontal (x direction) velocity is unaffected.

After two checks, the Spectrum rubs out the old ball by overprinting it with a blank space (Line 19Ø) and GOes back TO Line 8Ø to carry on.

```
5 S=4Ø:CC=55296:D=21:
   SC=1Ø24:A1=18:A2=1Ø
1Ø FOR Z=Ø TO S-1:PRINT"█ ▩";:
   R$=R$+"▐":S$=S$+"□":
   NEXT Z
2Ø FOR Z=Ø TO D:PRINT"▩ ▟ π"
   LEFT$(S$,S-2)"█ ▄ ▩";:
   D$=D$+"▜":NEXT Z
3Ø FOR Z=Ø TO S-1:PRINT"▩";:
   NEXT Z
5Ø FOR Z=1 TO S:X=(INT(RND(1)
   *A1)+1)*2:Y=(INT(RND(1)
   *A2)+1)*2
6Ø PRINT "▤ ▜"LEFT$(D$,Y)
   LEFT$(R$,X)"▟ □":NEXT
7Ø X=1Ø:Y=1Ø:XV=-1:YV=1
8Ø POKE SC+X+Y*S,215:OY=Y:
   OX=X
9Ø X=X+XV:Y=Y+YV
1ØØ C1=PEEK(CC+(X-1)+Y*S)
   AND 15
1Ø5 C2=PEEK(CC+X+(Y-1)*S)
   AND 15
11Ø C3=PEEK(CC+(X+1)+Y*S)
   AND 15
115 C4=PEEK(CC+X+(Y+1)*S)
   AND 15
12Ø IF C1=4 OR C3=4 THEN XV=-XV
125 IF C2=4 OR C4=4 THEN YV=-YV
13Ø POKE SC+OX+OY*S,16Ø:
   GOTO 8Ø
```

Enter Lines 1Ø to 13Ø from the Commodore 64 program then add:

```
5 S=22: CC=384ØØ: D=19:
   SC=768Ø: A1=9: A2=8
```

The Commodore programs start by setting up a border and a series of randomly positioned squares, all of which are the same colour (purple). The ball is a different colour, as of course is the background colour.

Line 5 sets up the variables for the program. S is the number of characters in a line (a variable is used for this since the Commodore 64 and the Vic have different-sized screens); CC is the start address of the screen colour memory, which is again different for the Vic and 64, as is the start address of the screen memory, represented by variable SC. The last two variables, A1 and A2, are the limits within which the computer can put the random blocks. These are set within a smaller area than the border to save any of the blocks being wasted by putting them at the same position as the border.

Lines 2Ø to 6Ø put the border in, and add the random squares. The computer next moves the cursor, and sets up four new variables (x and y for the position of the ball and xv and xy for the initial speed in each direction). Line 8Ø POKEs the ball onto the screen. It does this rather than PRINTing the ball, since PRINTing it would need more cursor control characters, and this would slow the program down: the computer would have first to move the cursor, and then PRINT the ball.

This POKE command actually puts the value 215 into whichever screen location the program calculates that the ball should be in. 215 is the ASCII code of the ball character.

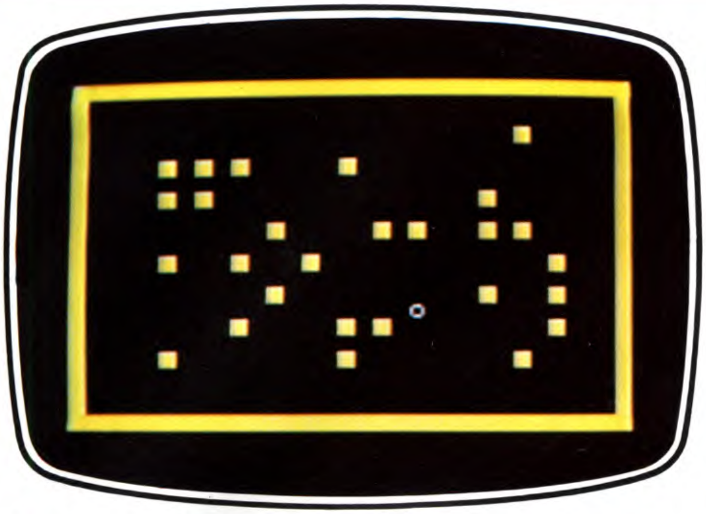
Line 9Ø adds the x (xv) speed to the x coordinate, and the y (yv) speed to the y coordinate, to give the new position of the ball.

CHECKING THE COLOUR

The next four lines set up four variables, C1 to C4, equal to the number of the colour in the four squares surrounding the ball. After that, there are two program lines which check on the colour; colour 4 is the colour of both the border and the random squares.

If a 4 is found in any of the four squares surrounding the ball, the next part of the program will instruct the ball to reverse in the direction that it would hit—that is, move away from the obstacle. Two separate checks are made, one in the horizontal (x) direction, one in the vertical (y). Note that it is not simply a matter of reversing the ball in both directions. To do so would simply make it back-track.

The two x checks, or the checks on the squares to the right and the left of the ball, are both made in the same IF . . . THEN statement.



The Spectrum displays random blocks



The Dragon's colourful pattern

This is possible because the result is the same if either is true. The x speed is reversed by changing its sign—either to or from negative, so that −5 becomes 5, and 5 becomes −5. The same is also true of the y checks.

Then the last line of the program POKEs 16Ø into the old position of the ball (held by the variables ox and oy) to blank out the old ball. The program then GOes back TO line 8Ø to carry on moving the ball.

```
10 X=30:Y=30:VX=4:VY=4
20 MODE2
30 MOVEX,0:PLOT85,0,1023:PLOT85,
   X,1023:MOVEX,1023-Y:PLOT85,
   1280,1023:PLOT85,1280,1023-Y:
   MOVE1280-X,1023-Y:PLOT85,
   1280,0:PLOT85,1280-X,0:MOVE
   1280-X,Y:PLOT85,0,0:PLOT85,0,Y
40 PROCOBJS:GCOL3,3:X=600:Y=500:
   GOTO 70
50 X2=X:Y2=Y:X=X+VX:Y=Y+VY
60 PLOT69,X2,Y2
70 PLOT69,X,Y
80 IF POINT(X+VX*2,Y)<>0 THEN
   VX=-VX:SOUND1,-15,100,1
85 IF POINT(X,Y+VY*2)<>0 THEN
   VY=-VY:SOUND1,-15,100,1
90 IF POINT(X+VX*2,Y+VY*2)<>0
   THEN VX=-VX:VY=-VY:
   SOUND1,-15,100,1
100 GOTO 50
110 DEF PROCOBJS
120 FOR T=0 TO 3
130 FOR P=1 TO 15:GCOL0,RND(3)
140 X=RND(500)+50+640*(T AND 1):
    X2=X+RND(100):Y=RND(400)+
    50+250*(T AND 2):Y2=Y+RND(90)
150 PROCBLOCK(X,Y,X2,Y2):NEXT:
    NEXT:ENDPROC
160 DEF PROCBLOCK(A,B,C,D)
170 MOVEA,B:MOVEA,D:PLOT85,C,B:
    PLOT85,C,D:ENDPROC
```

The Acorn program starts off by setting up four variables, and selecting MODE 2. The four variables are X and Y (first used to set the thickness of the border around the screen edge, and then used as the ball's coordinates), and VX and VY. VX and VY are the initial speed vectors of the ball, representing the velocity the ball is going in one direction, here either the x direction or the y direction.

Then, in Line 3Ø, the computer draws in the border, using the variables X and Y. Line 4Ø sends the computer to PROCOBJS, which adds the coloured blocks to the screen.

PROCOBJS splits the screen into four rectangles, leaving a narrow cross in the centre of the screen (to ensure that the ball has a clear path when it first moves off). The FOR ... NEXT loop in Line 12Ø makes the computer run through the following few lines four times, so that the blocks appear in each of the four rectangles.

The colour of each block is chosen randomly, as is the position and size of each small block inside the main rectangles. (In fact, the computer calls PROCBLOCK from within PROCOBJS to draw each small block.)

When this PROCedure has been finished, the computer returns to Line 4Ø, where it alters the variables X and Y so that they are equal to the ball's starting coordinates, 6ØØ, 5ØØ.

Lines 5Ø and 6Ø rub out the old ball, using variables X2 and Y2—and because there is no 'old position' to rub out when the computer runs through the program for the first time, Line 4Ø also tells the computer to GOTO Line 7Ø.

This prints the ball in its new position, before the computer checks the colours of the three pixels in front of the ball.

You can work out how POINT works from these check lines: 8Ø, 85, 9Ø. As you can see, it takes this form:

POINT (x coordinate, y coordinate)

As with the other, similar functions, you can't use it directly—you must precede it with a command such as PRINT, or, as in this program, IF.

At least one of the two numbers in each of these checks is a calculation involving VX or VY. The reason for this is so that the computer

only makes its checks in the direction the ball is moving (any other direction does not matter). In fact it checks the three pixels diagonally in front of the ball—the one in front and the ones to the right and left of it. Thus the computer only needs to check three pixels, instead of several more on all sides of the ball.

If the colour of any of the three pixels is not black (colour Ø) the computer makes a short beep, and reverses the relevant speed vector (or vectors). Reversing the speed vector changes the ball's direction, thus 'bouncing' the ball away from the object it has just hit. Note that the checks mean that the ball's speed will only be reversed in one direction. It is not a matter simply of altering its speed in both directions, which would make the ball back-track, rather than reflect away from the obstacle.

After this, the computer GOes back TO Line 5Ø, to rub out the ball in its old position, and reprint it in its new position. Line 5Ø also updates the ball's coordinates by the speed vector VX and VY. Then the computer runs through the same sequence again.

```
10 CLS0
20 PRINTSTRING$(32,223);:PRINT
   @448,STRING$(32,223);
30 FORK=1TO7:PRINT@32*K,CHR$
   (223)+STRING$(7-K,191);
40 PRINT @32*K+24+K,STRING$
   (7-K,175)+CHR$(223);
50 PRINT@448-32*K,CHR$(223)
   +STRING$(7-K,175);:PRINT
   @448-32*K+24+K,STRING$
   (7-K,191)+CHR$(223);
60 NEXT
70 FORK=1TO16:READA,B:
   POKE1024+A,B:NEXT
80 DATA 137,143,114,159,111,159,112,
   159,113,159,150,255,118,255,203,
   239,276,239,296,255,264,255,366,
   159,367,159,368,159,365,159,406,143
90 X=32:Y=16:VX=2-RND(39)/10:
   VY=2-RND(39)/10
100 SET(X,Y,5)
110 P1=POINT(X+VX,Y+VY):P2=
   POINT(X+VX,Y):P3=POINT
   (X,Y+VY):IF P1=0 AND P2=0 AND
   P3=0 THEN 190
120 IF P2=0 AND P3=0 THEN 160
130 IF P2>0 THEN VX=-VX:IF P2<4
   THEN VX=VX-SGN(VX)*RND(0) ELSEIF
   P2>3 THEN VX=VX+SGN(VX)*RND(0)
140 IF P3>0 THEN VY=-VY:IF P3<4
   THEN VY=VY-SGN(VY)*RND(0) ELSEIF
   P2>3 THEN VY=VY+SGN(VY)*RND(0)
150 GOTO 190
160 VX=-VX:VY=-VY
170 IF P1>3 THEN VX=VX+SGN(VX)
   *RND(0):VY=VY+SGN(VY)*RND(0)
180 IF P1<4 THEN VX=VX-SGN(VX)
   *RND(0):VY=VY-SGN(VY)*RND(0)
190 IF ABS(VX)<1 THEN VX=SGN(VX)
200 IF ABS(VX)>2 THEN VX=2*SGN(VX)
210 IF ABS(VY)<1 THEN VY=SGN(VY)
220 IF ABS(VY)>2 THEN VY=2*SGN(VY)
230 RESET(X,Y):X=X+VX:Y=Y+VY
240 GOTO 100
```

The Dragon and Tandy program offers an example of how useful POINT can be. The screen display is slightly different from those on the other computers, as you can see from the screen pictures.

This screen display does not have randomly placed blocks, but has diagonal coloured corners. Trying to use variables to test whether or not the ball is hitting these corners would be not only very complicated, but also very tedious. And it would make the program very slow. Even when you had done this, you would still have to add several more IF ... THEN statements to check whether or not the ball was hitting any of the six blocks in the middle of the screen. Clearly, this would not be practical.

CHECKS FOR SEVERAL COLOURS

Using POINT to detect the colour of three squares around the ball needs only a few program lines. And by choosing the colours carefully, you can check for several different colours and results with just two checks.

The program first sets up the background: it clears the screen with a black background, and puts in the corners, two red and two blue, in Lines 1Ø to 6Ø. Line 7Ø READs the DATA from Line 8Ø to POKE the remaining blocks onto the screen. It POKEs them rather than PRINTing them because it saves about four program lines by READing DATA.

Variables for the initial coordinates of the ball (X and Y) and for the initial x and y speeds (the components of the speed along the x axis and the y axis are represented by VX and VY) are set up by Line 9Ø. The computer then SETs a low resolution 'pixel' at the x and y position for the ball.

The colours are checked in two stages. First, three variables are set up. These are for the square diagonally in front of the ball, whichever direction the ball is travelling in, and for the next x and the next y squares. The variables are P1, P2, and P3. Since it is not necessary to check squares in directions where the ball is not moving, these variables will allow the program to make only the three important checks.

USING POINT

You can see how the POINT command works from Line 11Ø. It takes the general form:

POINT(X coordinate, Y coordinate)

The same Line, 11Ø, also checks to see whether each of the three squares are in colour Ø. Colour Ø is black, which is the background colour—if all three colours are black, then the ball is not hitting anything, and so the computer GOes TO Line 19Ø, missing out the reflection checks.

If the next x (horizontal direction) and the next y (vertical direction) squares are both black, but the diagonally-next square is not, the computer jumps to Line 16Ø. This is because the ball needs to bounce back in precisely the same direction from which it has just come, and Line 16Ø is designed to cope with the special case of a diagonal collision. Both of the speed vectors (the speed in the x direction and in the y direction) are set equal to minus whatever they were before, making the ball back-track.



The following lines also change the speed, according to what colour the ball is hitting (see below).

If neither of the checks carried out by the computer up to Line 13Ø are true, the ball must be hitting an object (either a corner or a block) from the side. Line 13Ø checks whether it is the next square horizontally which is being hit, and if so does several things. First it alters the relevant speed vector. The 'relevant' vector is the opposite to the axis of the collision, so in this example the y vector is changed. It then checks to see which limits the colour of the obstructing square falls into.

SPEEDING UP THE BALL

The colours 1, 2 and 3 are green, yellow, and blue. Whenever the colour is one of these, the speed is reduced by a small randomly set amount. If the colour's number is greater than 3, then the square must be either red, buff, cyan, magenta or orange. When this is the case, the ball is speeded up by a small, random, amount.

The advantages of having a small number of results IF the colour is, say, blue, and of choosing colours carefully are now exposed. By using colours with numbers next to each other to yield the same result, you can reduce the number of IF ... THEN checks you need to use, saving memory, and making your program faster.

This program, for example, uses colours 1 to 3 to slow the ball down, and colours 4 to 8 to speed it up.

Line 14Ø does the same as Line 13Ø, except that it checks the vertical values, not the horizontal ones. The computer then jumps to Line 19Ø. The routine which starts here checks the value of the speed. If the speed gets greater than 2, the program would begin to look as if it had some bugs in it. Since the computer is missing out a number of squares to simulate the fast movement, this would cause strange things to happen with the tests for the collisions (the ball might pass through a block or corner, if this happened to be a square that it missed, for example).

For this reason, the check ensures that speed is kept to less than 2. The same sort of problem would occur with a speed of less than 1, so a further check ensures that if the speed is this low, it is altered to prevent any odd effects. Note that all of the set values are ABSolute values—ignoring a plus or minus sign. Although −1 is actually less than 1, the magnitude of the speed is the same, and the minus sign only indicates that it is in the opposite direction. The ABS just gets rid of the sign before checking the speed value.

The next Line, 23Ø, RESETs the block (the ball) to the background colour, to rub out the ball from its last position. Then the program updates the ball's speed, and sends the computer back to Line 1ØØ to move the ball again.

You could quite simply add some lines to the programs above to convert them into games. The Dragon and Tandy program, especially, could be the basis for a pinball type of game. You can get the computer to do almost anything after the IF ... THEN conditions, so that you could add sound effects and scoring very easily. Future articles in *INPUT* will use the same commands in a wide variety of games routines.

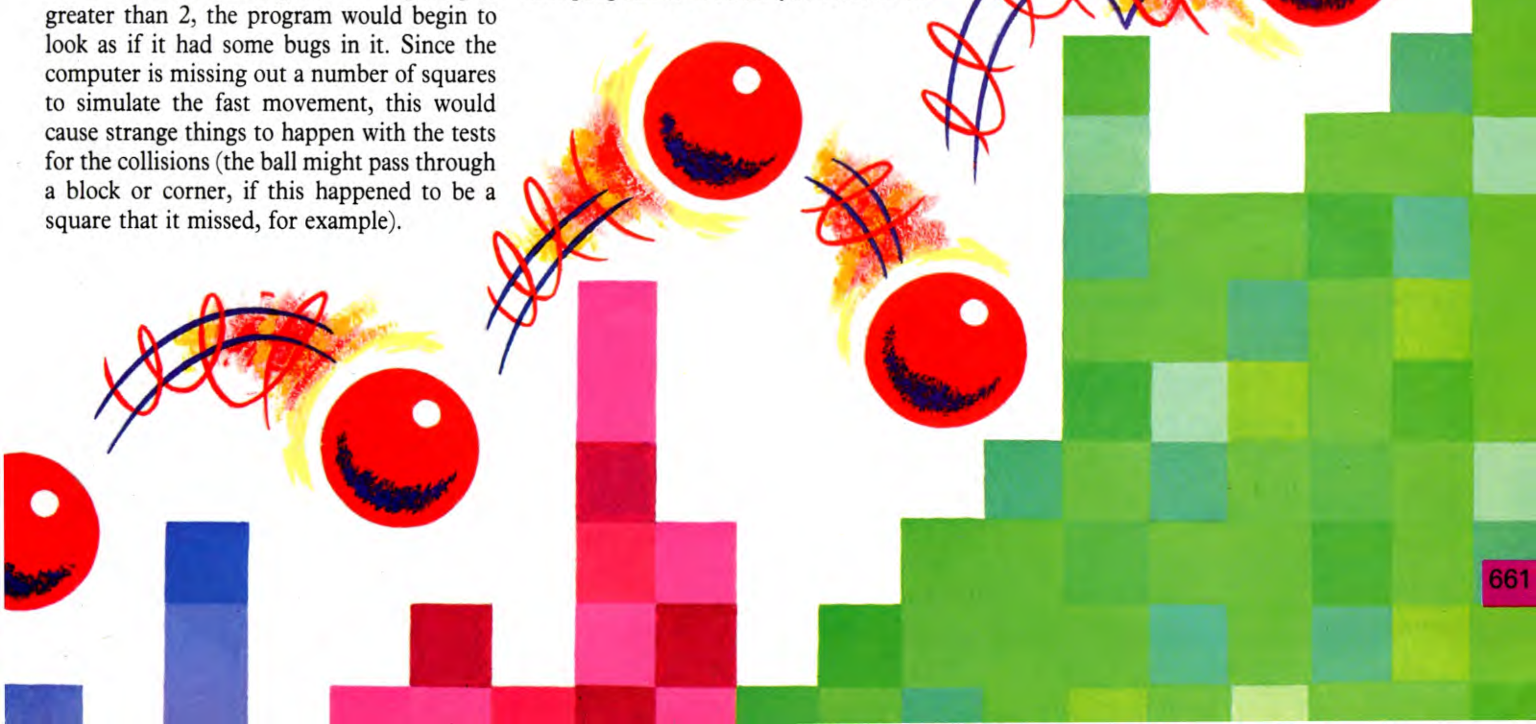
The Dragon and Tandy are not just confined to POINT, which is used on the low resolution graphics screen. There is another command, PPOINT, which you can use on the high resolution screen.

Instead of giving the colour of a particular block or character square, as POINT, PPOINT gives the colour of a pixel. Type in and RUN this program to see how you can use it.

```
10 PMODE 1,1
20 PCLS3
30 SCREEN1,0
40 LINE(20,20)-(235,171),
   PRESET,BF
50 VX=RND(7)-4:VY=RND(7)-4
60 BX=127:BY=95
70 PSET(BX,BY,4)
80 IF PPOINT(VX+BX,BY)=3 THEN
   VX=RND(3)*((VX>0)-(VX<0))
90 IF PPOINT(BX,BY+VY)=3 THEN
   VY=RND(3)*((VY>0)-(VY<0))
100 PRESET(BX,BY)
110 BX=BX+VX:BY=BY+VY
120 GOTO 70
```

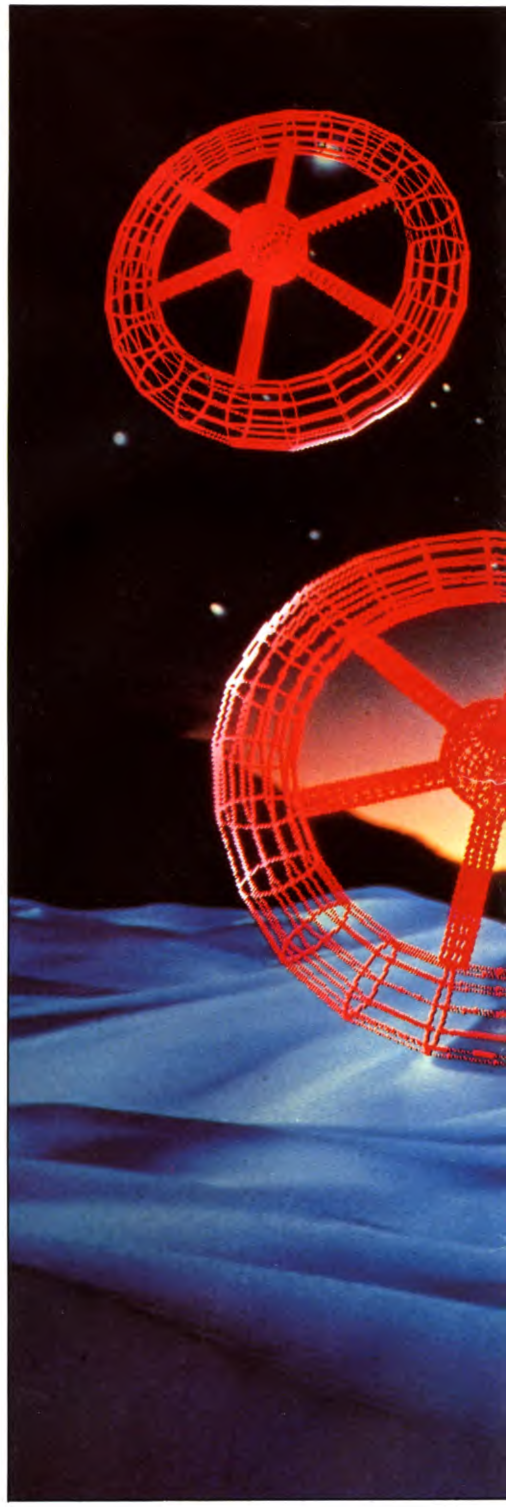
A ball bounces around the screen. PPOINT is used to check if the ball is in contact with the blue cushion. Lines 8Ø and 9Ø check if the pixel ahead of the ball is blue—colour number 3. If that pixel is blue then the ball is put into reverse by the remainder of the two lines, working out a random distance for the ball to move.

You can use PPOINT for collision detection generally. If you have one graphic bouncing off another, for example, you can test if they are in contact by checking for the colour of one of the graphics.



WIREFRAMES - ADDING CURVES

The earlier articles on wireframe drawing provide an impressive battery of standard routines. All you need now is suitable calls to combine these into new images



So far, you've seen how to set up basic shapes for rectangular grids and concentric circles. And with various transformations, you were able to combine a series of grids into perspective views of a cube. This article makes minor changes to the program developed so far, to extend the range of wireframe images you can draw.

To start with, it is very easy to set up multiple images. As you put the program starting on page 606 through its paces, you will have noticed some spectacular perspective views when D (the viewing distance) is small—100 to 400, say, whereas there is little perspective effect when D is of the order of thousands. You can continue this exercise so it looks as if you see not one, but four cubes on the screen at the same time.

MULTIPLE OBJECTS

If you SAVEd the program from the previous article (pages 605 to 611) you can spare yourself a lot of typing by LOADing it back into the computer. If you did not SAVE a copy of the program, key the lines again. Also make sure you have the Circle routine, which appeared in the first article (page 513).

Since all the programming is written in self-contained blocks or routines, it is simple to modify it to draw multiple images, instead of just one. To draw four cubes, for example, you need to specify their screen position and call the Cube routine four times. The position is best specified in the routine to transform the coordinates. Modify this part of the listing as follows, but do not RUN the program yet:

```
8500 LET X1=T1*X+T4*Y+T7+XO
8510 LET Y1=T2*X+T5*Y+T8+YO
8520 LET Z1=T3*X+T6*Y+T9+ZO
```

```
8500 X1=T1*X+T4*Y+T7+XO
8510 Y1=T2*X+T5*Y+T8+YO
8520 Z1=T3*X+T6*Y+T9+ZO
```

```
8530 X1=T11*X+T12*Y+T13+XO
8540 Y1=T21*X+T22*Y+T23+YO
8550 Z1=T31*X+T32*Y+T33+ZO
```

The variables XO, YO and ZO at the end of these lines specify an offset in each of the three coordinate directions to determine the separation of the four cubes. These variables are given values just before each cube is drawn.

Now enter the next section of program and RUN it to see the effect of the offset:

```
120 LET L=20: LET PP=20: LET N=1
150 GOSUB 1500
1500 LET XO=-PP: LET YO=-PP: LET
  ZO=0
1520 GOSUB 1000
1530 LET XO=PP: LET YO=-PP: LET
  ZO=0
1540 GOSUB 1000
1550 LET XO=PP: LET YO=PP: LET ZO=0
1560 GOSUB 1000
1570 LET XO=-PP: LET YO=PP: LET
  ZO=0
1580 GOSUB 1000
1590 RETURN
```

```
120 L=20:PP=20:N=2
150 GOSUB 1500
1500 XO=-PP:YO=-PP:ZO=0
1520 GOSUB 1000
1530 XO=PP:YO=-PP:ZO=0
1540 GOSUB 1000
1550 XO=PP:YO=PP:ZO=0
1560 GOSUB 1000
1570 XO=-PP:YO=PP:ZO=0
1580 GOSUB 1000
1590 RETURN
```

The Vic 20 version of this section of the listing is as for the Commodore 64, except the following line:

```
120 L=50:PP=80:N=1
```

```
120 L=50:P=100:N=5
150 IF V THEN PROCCUBES
1500 DEF PROCCUBES
1510 PROCOFFSET(-P,-P,0)
1520 PROCCUBE
```



```
1530 PROCOFFSET(P,-P,Ø)
1540 PROCCUBE
1550 PROCOFFSET(P,P,Ø)
1560 PROCCUBE
1570 PROCOFFSET(-P,P,Ø)
1580 PROCCUBE
1590 ENDPROC
8800 DEF PROCOFFSET(X,Y,Z)
8810 XO=X
8820 YO=Y
8830 ZO=Z
8840 ENDPROC
9080 PROCOFFSET(Ø,Ø,Ø)
```

The Dragon and Tandy versions of this section of the listing are as for the Commodore 64, except the following line:

```
120 L=10:PP=20:N=2
```

Line 12Ø has been changed to include the variable PP (P on the Acorns), which passes a value for the offset at Line 15ØØ. (On the Acorns, this value is passed to a routine between Lines 88ØØ and 884Ø. Line 9Ø8Ø sets each of the variables to Ø in the Initialization routine.)

Line 15Ø calls a routine to draw four cubes. The subroutine merely specifies a position (as in Line 153Ø, for example), then calls the Cube routine at the next line to draw the four cubes.

When you RUN the program, respond to the prompts on the screen. The first prompt is for you to enter the value of D—the projection plane distance. A value of 1ØØØ is good to begin with. The next prompt is for the eye position, so enter values for the X, Y and Z positions. Below are some values that will give images on the screen. Try them and note how the images are distorted when D is small, but not when it is large. Try other values of your own, including several thousand for D, and negative or zero values for X, Y and Z.

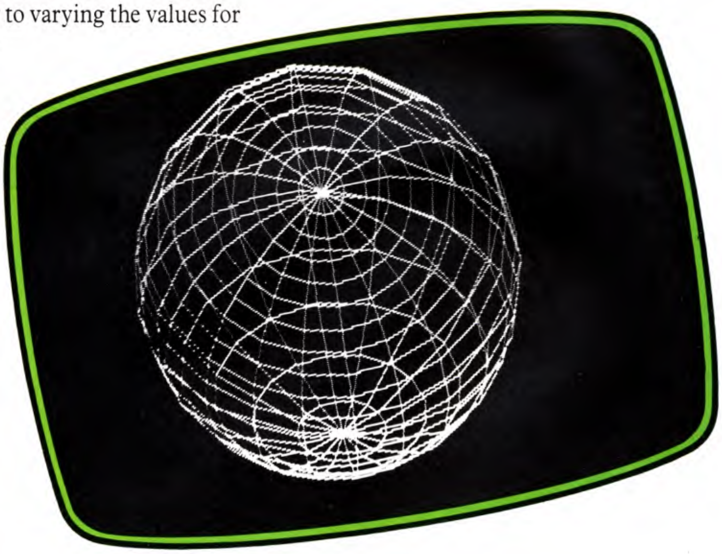
D	X	Y	Z
1ØØ	55	Ø	Ø
9ØØ	2ØØ	-25Ø	2ØØ
2ØØØØ	1ØØØ	3ØØØ	1ØØØØ

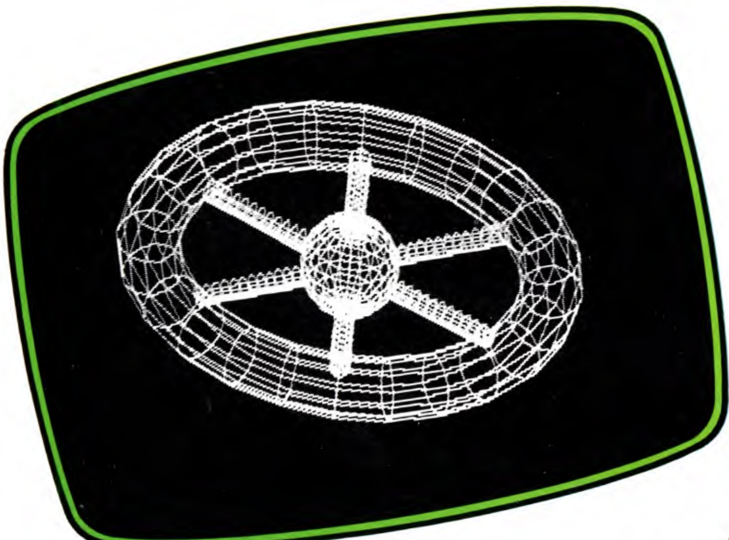
The wide variation of views possible with this program should give you plenty of scope for interesting experiments, but do not restrict yourself merely to varying the values for eye position. Notice that Line 12Ø in the last section of program you entered variables for the set length of sides of the cube (L), the separation of cubes from each other (PP or P) and the number of grid squares (N) in each side—three other variables that you can change before RUNning the program.

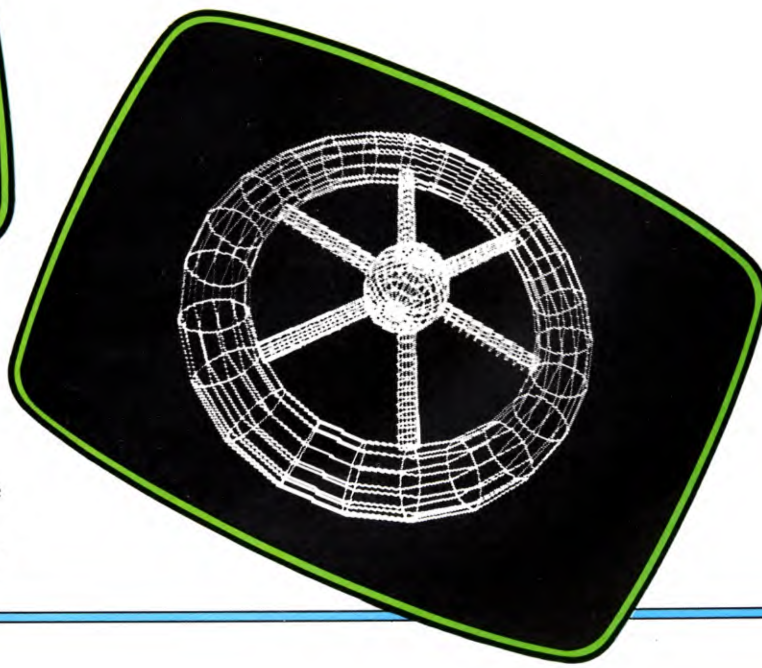
DRAWING A GLOBE

The program you have just entered is a short one, but it is able to make a complex image because it can call up any of the many routines that make up the rest of the listing. One of these routines should be the one to draw concentric circles, which was listed on page 513. If your program does not include this routine, key it now then SAVE a copy of the whole program for use in the future.

You can now modify the program so that it calls the Circle routine, instead of those for a grid, side and cube. Delete Line 12Ø, then enter the following lines:

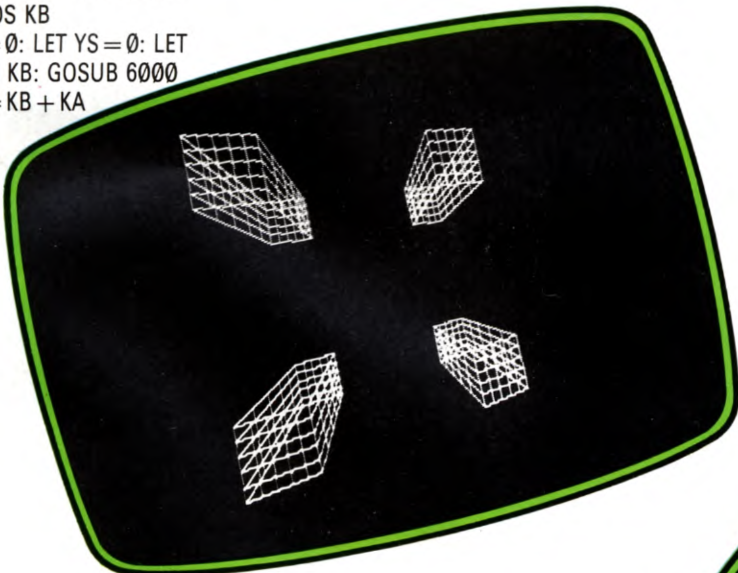






When the viewing distance is long, the image shows little or no perspective, but on reduction, distant points are drawn smaller. The globe was drawn from a distant viewpoint, but the cubes were drawn from much closer

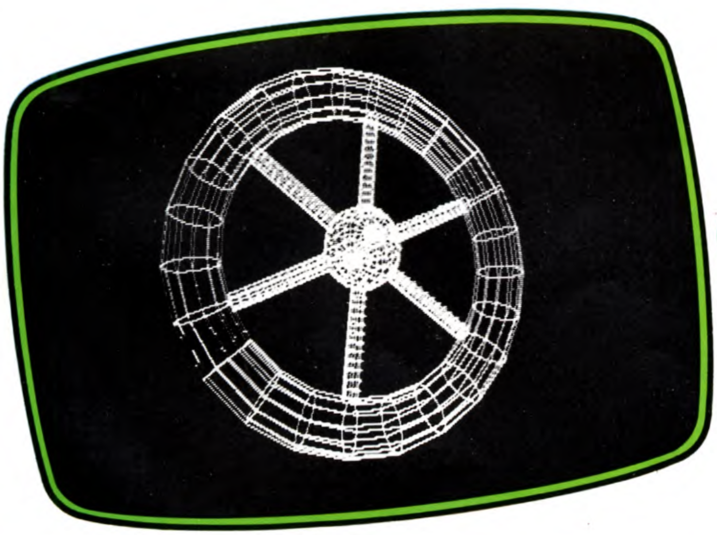
```
100 LET XO=Ø:LET YO=Ø:LET ZO=Ø
150 LET R=2Ø: LET N=18: GOSUB 2ØØØ
2ØØØ LET T7=Ø: LET T8=Ø: LET T9=Ø
2Ø1Ø LET T4=Ø: LET T5=Ø: LET T6=1
2Ø4Ø LET KA=2*PI/N
2Ø5Ø LET KB=Ø
2Ø6Ø FOR K=1 TO INT (N/2)
2Ø7Ø LET T1=COS KB: LET T2=SIN KB: LET
   T3=Ø
2Ø8Ø LET XS=Ø: LET YS=Ø: GOSUB 6ØØØ
2Ø9Ø LET KB=KB+KA
21ØØ NEXT K
211Ø LET T1=1: LET T2=Ø: LET T3=Ø
212Ø LET T4=Ø: LET T5=1: LET T6=Ø
213Ø LET KA=KA/2
214Ø LET KB=Ø: LET RR=R
215Ø FOR K=1 TO N
216Ø LET T7=Ø: LET T8=Ø: LET
   T9=RR*COS KB
217Ø LET XS=Ø: LET YS=Ø: LET
   R=RR*SIN KB: GOSUB 6ØØØ
218Ø LET KB=KB+KA
219Ø NEXT K
22ØØ RETURN
```

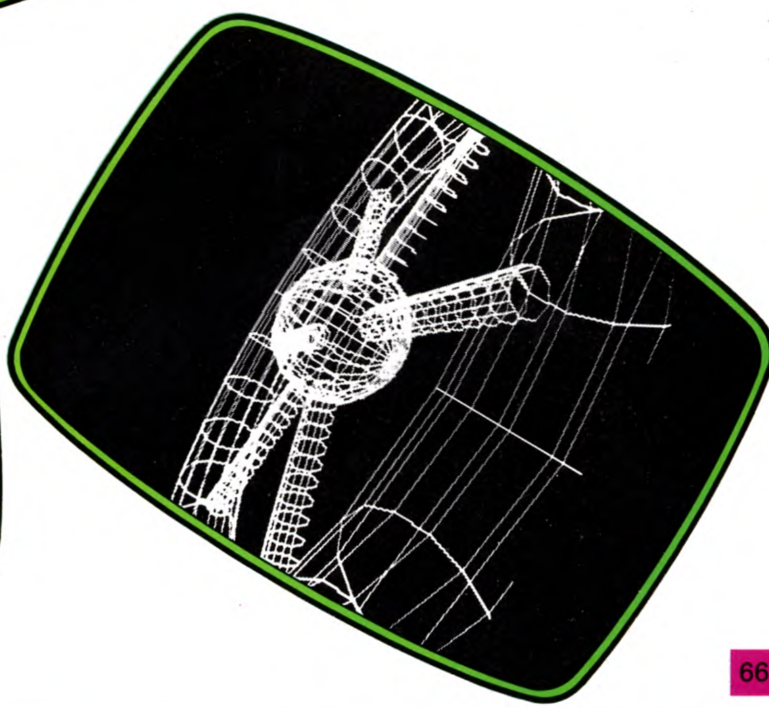


```
15Ø R=2Ø:N=18:GOSUB2ØØØ
2ØØØ T7=Ø:T8=Ø:T9=Ø
2Ø1Ø T4=Ø:T5=Ø:T6=1
2Ø4Ø KA=2*PI/N
2Ø5Ø KB=Ø
2Ø6Ø FORKC=1 TO INT(N/2)
2Ø7Ø T1=COS(KB):T2=SIN(KB):T3=Ø
2Ø8Ø XS=Ø:YS=Ø:GOSUB6ØØØ
2Ø9Ø KB=KB+KA
21ØØ NEXT KC
211Ø T1=1:T2=Ø:T3=Ø
212Ø T4=Ø:T5=1:T6=Ø
213Ø KA=KA/2
214Ø KB=Ø:RR=R
215Ø FOR KC=1 TO N
216Ø T7=Ø:T8=Ø:T9=RR*COS(KB)
217Ø XS=Ø:YS=Ø:R=RR*SIN(KB):
   GOSUB6ØØØ
218Ø KB=KB+KA
219Ø NEXT KC
22ØØ RETURN
```

```
15Ø IF V THEN PROCGLOBE(1ØØ,18)
2ØØØ DEF PROCGLOBE(R,N)
2Ø1Ø LOCAL KA,KB,KC
2Ø2Ø PROCorigin(Ø,Ø,Ø)
2Ø3Ø PROCYvector(Ø,Ø,1)
2Ø4Ø KA=2*PI/N
2Ø5Ø KB=Ø
2Ø6Ø FOR KC=1 TO N DIV 2
2Ø7Ø PROCXvector(COS(KB),SIN
   (KB),Ø)
2Ø8Ø PROCCIRCLE(Ø,Ø,R,N)
2Ø9Ø KB=KB+KA
21ØØ NEXT KC
211Ø PROCXvector(1,Ø,Ø)
212Ø PROCYvector(Ø,1,Ø)
213Ø KA=KA/2
214Ø KB=Ø
215Ø FOR KC=1 TO N
216Ø PROCorigin(Ø,Ø,R*COS(KB))
217Ø PROCCIRCLE(Ø,Ø,R*SIN(KB),N)
218Ø KB=KB+KA
219Ø NEXT KC
22ØØ ENDPROC
```

When you RUN this program, you should go through the familiar input stage which lets you select a value for D, and then one for the





Q+A

Is there a simple way to speed up the drawing of the space station?

The most radical way to speed up the program is to reduce the number of steps in the basic routines—the Circle routine for the space station and the Grid routine for the cubes. This method, however, will reduce the smoothness of ellipses and spoil the structure of the shape. A better way is to reduce the number of times these routines are called.

For example, you could have fewer sections across and along the tube of a torus, by reducing the values of R, N, RT (R2 for Acorns) and N2 in the Torus routine starting at Line 2ØØØ. At the same time, you must change the program that calls the routine so that the circles or ellipses are spaced out evenly and the program branches when the last of a family of curves is drawn, instead of going through the motions of drawing for longer than necessary.

Perhaps the easiest way to speed up execution is to abort the drawing of certain sections (like the spokes of the station). This is easily done with GOTO statements.

eye position. Using these values, Line 15Ø calls the routine you have just keyed to draw a picture of a globe. This is made up from a series of ellipses (or circles, depending on the eye position). These form the lines of longitude (passing through the poles of the globe) and latitude, which run parallel with the equator.

The radius of the globe is set by the variable R (Line 15Ø, except for the Acorns), and the number of lines of longitude and latitude is set by N. On the Acorn micros, these values are passed from the procedure at Line 15Ø to the variables at Line 2ØØØ. On all the computers, N should be an even number greater than 2. Lines 2ØØØ to 2Ø5Ø set up variables to position the globe and for use as counters. The loop between 2Ø6Ø and 21ØØ draws the lines of longitude and the one between 215Ø and 219Ø draws the lines of latitude.

The axis of the globe runs through the poles, and its direction is determined by the variables you INPUT. To see how the poles of the globe are positioned in relation to the screen axes, enter Ø as two of the coordinates of the eye position (Ø,Ø,2ØØ for example), and you should see an end-on view of the globe. Now you can enter positive or negative values for all three coordinates to see how the poles shift relative to the axes.

Users of the Vic 20 might find that their machine is running short of memory, despite the Super Expander cartridge. This problem can be solved by deleting the routines for one and four cubes. These are Lines 1ØØØ to 159Ø (the cubes) and Lines 5ØØØ to 513Ø (the grid).

FROM GLOBE TO TORUS

A globe is one of the simplest shapes that can be represented by a series of ellipses, but the same program has all the elements necessary to draw a much more exotic shape—a torus (or doughnut, complete with a hole in the middle). Only minor modifications to the Globe routine are required, as below:

```
15Ø LET R=1Ø: LET N=18: LET RT=2Ø:
   LET N2=18: GOSUB 2ØØØ
2Ø4Ø LET KA=2*PI/N2: LET NC=N2
2Ø45 IF RT=Ø THEN LET NC=INT (NC/2)
2Ø6Ø FOR K=1 TO NC
2Ø8Ø LET XS=RT: LET YS=Ø: GOSUB.
   6ØØØ
213Ø LET KA=2*PI/N
2135 IF RT=Ø THEN LET KA=KA/2
217Ø LET XS=Ø: LET YS=Ø: LET
   R=RT+RR*SIN KB: LET N=N2:
   GOSUB 6ØØØ
```

```
15Ø R=1Ø:N=18:RT=2Ø:N2=18:
   GOSUB2ØØØ
2Ø4Ø KA=2*PI/N2:NC=N2
2Ø45 IF RT=Ø THEN NC=INT(NC/2)
2Ø6Ø FORKC=1 TO NC
2Ø8Ø XS=RT:YS=Ø:GOSUB6ØØØ
213Ø KA=2*PI/N
2135 IF RT=Ø THEN KA=KA/2
217Ø XS=Ø:YS=Ø:R=RT+RR*SIN
   (KB):N=N2:GOSUB6ØØØ
```

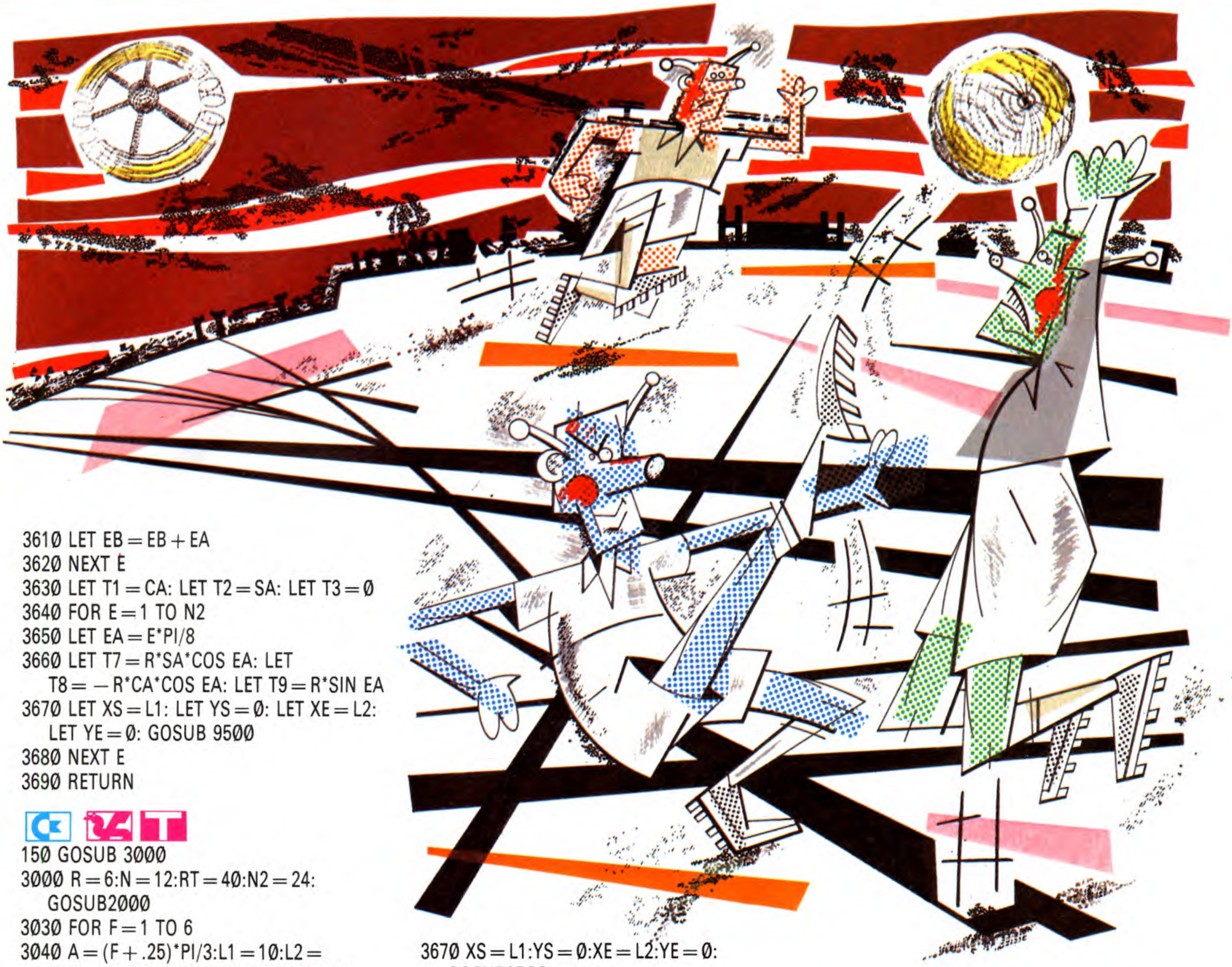
```
15Ø IF V THEN PROCTORUS(5Ø,18,
   1ØØ,18)
2ØØØ DEF PROCTORUS(R,N,R2,N2)
2Ø1Ø LOCAL KA,KB,KC,NC
2Ø4Ø KA=2*PI/N2:NC=N2
2Ø45 IF R2=Ø THEN NC=NC DIV 2
2Ø6Ø FOR KC=1 TO NC
2Ø8Ø PROCCIRCLE(R2,Ø,R,N)
213Ø KA=2*PI/N
2135 IF R2=Ø THEN KA=KA/2
217Ø PROCCIRCLE(Ø,Ø,R2+R*SIN
   (KB),N2)
```

RUN the program and give suitable values for D and the eye position, in response to the prompts on the screen. This time, R gives the internal radius of the tube, which has a number of segments round it, set by N. These segments are equivalent to the lines of longitude on the globe. RT (R2 for the Acorns) sets the distance (a radius) from the origin to the centre of the tube, and N2 gives the number of segments along the circumference of the tube. These segments are equivalent to the latitudes of the globe. The value of RT (or R2) should be greater than that of R, but in fact the routine works well even if this is not the case. Vary the values of these variables to see the effect. If RT (or R2) is zero and N is the same as N2, the torus degenerates into a sphere as drawn by the Globe routine.

COMBINING IMAGES

It would not be difficult to substitute the Globe or Torus routine for the Cube routine in the program that draws four images together on the screen. Equally simply, you could call any combination of all three routines (although on the Vic 20 you will be hampered by the memory limitations). You could, for example, draw cubes in a globe or a torus, or even a cube in a globe in a torus. And it would not be difficult to insert GOTO statements at suitable lines in the program to abort drawing the last two sides, so the image is less cluttered. Even Vic 20 users, however, can combine images to draw interesting shapes. Key these lines and RUN to see an impressive representation of a space station:

```
15Ø GOSUB 3ØØØ
3ØØØ LET R=6: LET N=24: LET RT=4Ø:
   LET N2=24: GOSUB 2ØØØ
3Ø3Ø FOR F=1 TO 6
3Ø4Ø LET A=(F+.25)*PI/3: LET L1=1Ø:
   LET L2=34: LET N1=12: LET R=2: LET
   N2=8: GOSUB 35ØØ
3Ø5Ø NEXT F
3Ø6Ø LET R=1Ø: LET N=12: LET RT=Ø:
   LET N2=12: GOSUB 2ØØØ
3Ø7Ø RETURN
35ØØ LET SA=SIN A
353Ø LET CA=COS A
354Ø LET T1=-SA: LET T2=CA: LET
   T3=Ø
355Ø LET T4=Ø: LET T5=Ø: LET T6=1
356Ø LET EA=(L2-L1)/N1
357Ø LET EB=L1
358Ø FOR E=Ø TO N1
359Ø LET T7=EB*CA: LET T8=EB*SA: LET
   T9=Ø
36ØØ LET XS=Ø: LET YS=Ø: LET N=N2:
   GOSUB 6ØØØ
```

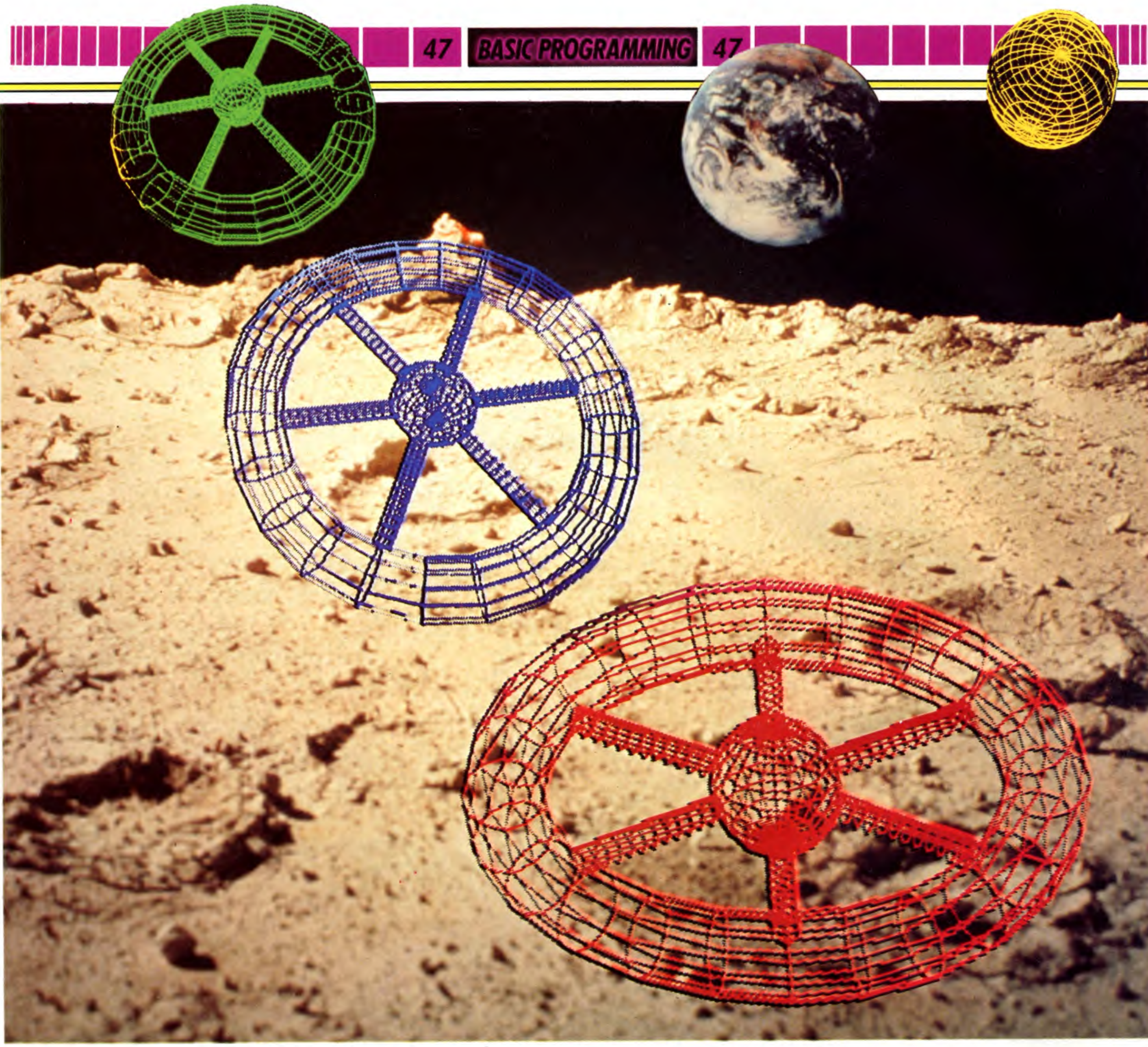


```
3610 LET EB=EB+EA
3620 NEXT E
3630 LET T1=CA: LET T2=SA: LET T3=Ø
3640 FOR E=1 TO N2
3650 LET EA=E*PI/8
3660 LET T7=R*SA*COS EA: LET
   T8=-R*CA*COS EA: LET T9=R*SIN EA
3670 LET XS=L1: LET YS=Ø: LET XE=L2:
   LET YE=Ø: GOSUB 95ØØ
3680 NEXT E
3690 RETURN
```

```
15Ø GOSUB 3ØØØ
3ØØØ R=6:N=12:RT=4Ø:N2=24:
   GOSUB2ØØØ
3Ø3Ø FOR F=1 TO 6
3Ø4Ø A=(F+.25)*PI/3:L1=1Ø:L2=
   34:N1=12:R=2:N2=8:GOSUB35ØØ
3Ø5Ø NEXT
3Ø6Ø R=1Ø:N=12:RT=Ø:N2=12:
   GOSUB2ØØØ
3Ø7Ø RETURN
35ØØ SA=SIN(A)
353Ø CA=COS(A)
354Ø T1=-SA:T2=CA:T3=Ø
355Ø T4=Ø:T5=Ø:T6=1
356Ø EA=(L2-L1)/N1
357Ø EB=L1
358Ø FOR EC=Ø TO N1
359Ø T7=EB*CA:T8=EB*SA:T9=Ø
36ØØ XS=Ø:YS=Ø:N=N2:GOSUB6ØØØ
361Ø EB=EB+EA
362Ø NEXT
363Ø T1=CA:T2=SA:T3=Ø
364Ø FOR EC=1 TO N2
365Ø EA=EC*PI/8
366Ø T7=R*SA*COS(EA):T8=-R*CA*
   COS(EA):T9=R*SIN(EA)
367Ø XS=L1:YS=Ø:XE=L2:YE=Ø:
   GOSUB95ØØ
368Ø NEXT
369Ø RETURN
```

```
15Ø GOSUB 3ØØØ
3ØØØ R=6:N=12:RT=4Ø:N2=24:
   GOSUB 2ØØØ
3Ø3Ø FOR F=1 TO 6
3Ø4Ø A=(F+.25)*PI/3:L1=1Ø:L2=
   34:N1=12:R=2:N2=8:
   GOSUB 35ØØ
3Ø5Ø NEXT
3Ø6Ø R=1Ø:N=12:RT=Ø:N2=12:
   GOSUB 2ØØØ
3Ø7Ø RETURN
35ØØ SA=SIN(A):CA=COS(A)
357Ø EB=L1
363Ø T1=CA:T2=SA:T3=Ø
364Ø FOREC=1TON2
365Ø EA=EC*PI/8
366Ø T7=R*SA*COS(EA):T8=-R*CA*
   COS(EA):T9=R*SIN(EA)
367Ø XS=L1:YS=Ø:XE=L2:YE=Ø:
   GOSUB 95ØØ
368Ø NEXT
369Ø RETURN
```

```
15Ø IF V THEN PROCSPACE
3ØØØ DEF PROCSPACE
3Ø1Ø LOCAL F
3Ø2Ø PROCTORUS(3Ø,12,2ØØ,24)
3Ø3Ø FOR F=1 TO 6
3Ø4Ø PROCSPOKE((F+Ø.25)*PI/3,5Ø,17Ø,
   12,1Ø,8)
3Ø5Ø NEXT F
3Ø6Ø PROCTORUS(5Ø,12,Ø,12)
3Ø7Ø ENDPROC
35ØØ DEF PROCSPOKE(A,L1,L2,N1,R,N2)
351Ø LOCAL EA,EB,EC,SA,CA
```



Microtip

Improving clarity

When the images of four cubes are small, as happens when the X, Y and Z values for the eye position are much larger than the viewing distance (D). it is difficult to see what is happening as the program goes through the drawing process. You can improve the clarity by inserting GOTO statements at suitable places in the program between Lines 1ØØØ and 117Ø. The simplest of these is GOTO 122Ø, inserted at Line 1125 (a new line) in the four cubes program.

```
352Ø SA=SIN(A)
353Ø CA=COS(A)
354Ø PROCXvector(-SA,CA,Ø)
355Ø PROCYvector(Ø,Ø,1)
356Ø EA=(L2-L1)/N1
357Ø EB=L1
358Ø FOR EC=Ø TO N1
359Ø PROCorigin(EB*CA,EB*SA,Ø)
36ØØ PROCCIRCLE(Ø,Ø,R,N2)
361Ø EB=EB+EA
362Ø NEXT EC
363Ø PROCXvector(CA,SA,Ø)
364Ø FOR EC=1 TO N2
365Ø EA=EC*PI/8
366Ø PROCorigin(R*SA*COS(EA),-R*
   CA*COS(EA),R*SIN(EA))
367Ø PROCLINE(L1,Ø,L2,Ø)
368Ø NEXT EC
369Ø ENDPROC
```

This program draws a torus, then six spokes radiating from its centre (these are altered on the Vic 20, due to lack of memory space). At the centre of these six spokes are a sphere, so that the result of the program is an image which looks like a model of a space station.

The scope for experiment with the torus program is virtually limitless. Try some small values (1ØØ to 5ØØ, say) for D, together with various values of X, Y and Z to give some interesting perspective views, especially with the near segment of the torus being clipped. Try also larger values for D and see how the distortion falls off.

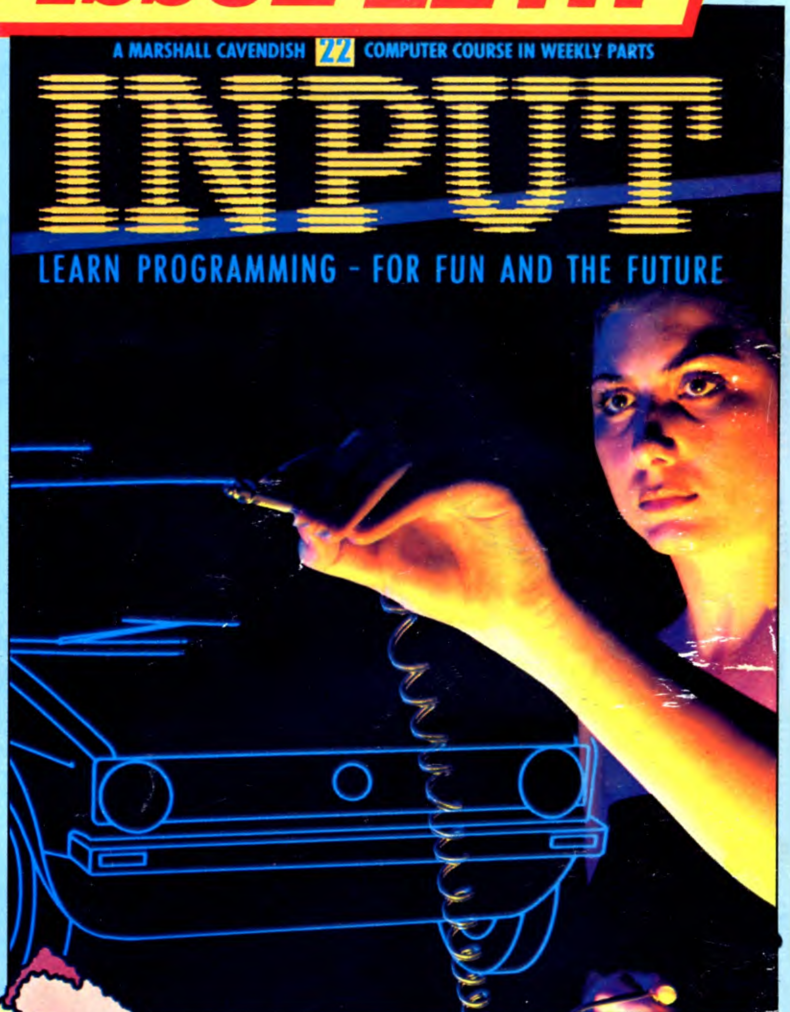
Now is the time to experiment with finishing touches, such as background and ink colour. If you're feeling really creative, you might also try to combine these routines with others to draw a rocket or night sky, for example.

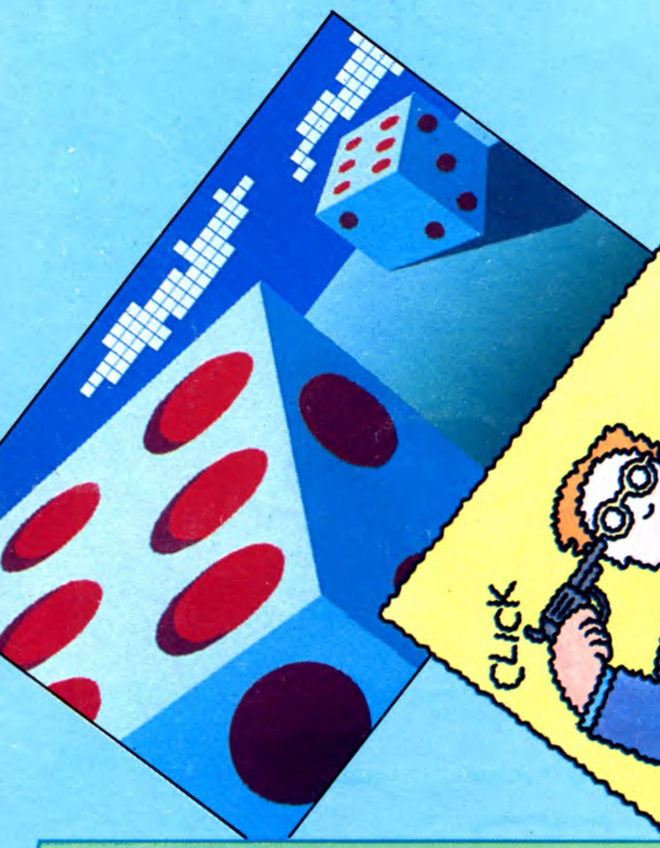
CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 22 . . .

- *Start to explore your computer's sound generator. The first of a series on MUSIC explains the theory and shows how to play TUNES ON THE KEYBOARD*
- *Learn about PROBABILITY, the fascinating rules that govern the outcome of events and form the basis of all kinds of games*
- *See how to USE YOUR TEXT COMPRESSOR in a typical adventure*
- *Find out what happens when you connect a LIGHT PEN to your micro*
- *For COMMODORE users, there's a machine code program that converts BASIC programs written for tape to give them DISK COMPATIBILITY*
- *PLUS . . . a short guide to CARING FOR TAPES AND DISKS*









ASK YOUR NEWSAGENT FOR INPUT